НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет прикладної математики

Кафедра програмного забезпечення комп'ютерних систем

«На правах рукопису»
УДК 004.7

«До захисту допущено»
Науковий керівник кафедри
_____ Іван ДИЧКА
«___»_____2020 р.

Магістерська дисертація

на здобуття ступеня магістра

за освітньо-професійною програмою
«Інженерія програмного забезпечення комп'ютерних
та інформаційно-пошукових систем»

зі спеціальності 121 Інженерія програмного забезпечення
на тему: «Алгоритмічно-програмний метод планування та динамічного виділення ресурсів для хмарних обчислень»

Виконав:
студент ІІ курсу, групи КП-91мп
Кривенко Петро Олегович _____

Керівник:
Ст. викл. кафедри ПЗКС, к.т.н.,
Хіцко Яна Володимирівна _____

Консультант з нормоконтролю:
Доцент кафедри ПЗКС, к.т.н., доцент
Онай Микола Володимирович _____

Рецензент:
Доцент кафедри СПіСКС, к.т.н., доцент,
Боярінова Юлія Євгенівна _____

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.

Студент _____

Київ – 2020 року

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет Прикладної математики
Кафедра Програмного забезпечення комп'ютерних систем

Рівень вищої освіти – другий (магістерський)
Спеціальність – 121 «Інженерія програмного забезпечення»
Освітньо-професійна програма «Інженерія програмного забезпечення комп'ютерних та інформаційно-пошукових систем»

ЗАТВЕРДЖУЮ
Науковий керівник кафедри
_____ Іван ДИЧКА
«___»_____ 2019 р.

ЗАВДАННЯ
на магістерську дисертацію студенту
Кривенку Петру Олеговичу

1. Тема дисертації «Алгоритмічно-програмний метод планування та динамічного виділення ресурсів для хмарних обчислень», науковий керівник дисертації Хіцко Яна Володимирівна, к.т.н., ст. викладач, затверджена наказом по університету від «12» листопада 2020 р. №3298-С.
2. Термін подання студентом дисертації «14» грудня 2020 р.
3. Об'єкт дослідження: процес планування задач та виділення ресурсів в хмарних обчисленнях.
4. Предмет дослідження: методи і алгоритми виділення ресурсів та планування задач у хмарних обчисленнях.
5. Перелік завдань, які потрібно розробити:
   - провести систематизацію методів і алгоритмів планування задач та виділення ресурсів в хмарних обчисленнях;
   - розробити порівняльну характеристику розповсюджених алгоритмів планування задач;
   - розробити модифікований метод динамічного виділення ресурсів в хмарних обчисленнях на основі алгоритмів Round Robin та Shortest Job First;
   - провести експериментальне дослідження ефективності створеного методу;
   - розробити хмарний сервіс, який реалізовує запропонований метод.
6. Перелік ілюстративного матеріалу:
   - блок-схема алгоритму;
   - схема структури системи;
   - схема взаємодії сервісів;
   - блок-схема обробки запиту;
   - діаграма отриманих результатів порівняння запропонованого методу;
   - канва бізнес-моделі.

7. Перелік публікацій:
   - Тези доповіді «Алгоритм динамічного виділення ресурсів для хмарних обчислень»
8. Консультанти розділів дисертації

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв
Нормоконтроль	Онай М.В., доцент кафедри ПЗКС		

9. Дата видачі завдання «11» жовтня 2019 р.

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів магістерської дисертації	Примітка
1.	Ґрунтовне ознайомлення з предметною галуззю	16.12.2019	
2.	Визначення структури магістерської дисертації; вивчення літератури, пошук додаткової літератури, патентний пошук	02.02.2020	
3.	Робота над першим розділом магістерської дисертації; проведення наукового дослідження	13.03.2020	
4.	Проведення наукового дослідження; робота над другим розділом магістерської дисертації; розроблення програмного забезпечення; підготовка матеріалів доповіді ПМК-2020	08.05.2020	
5.	Проведення наукового дослідження; робота над статтею за результатами наукового дослідження	14.08.2020	
6.	Проведення наукового дослідження; робота над третім розділом магістерської дисертації;	19.09.2020	
7.	Завершення роботи над основною частиною магістерської дисертації; підготовка ілюстративного матеріалу;	09.11.2020	
8.	Оформлення текстової і графічної частини магістерської дисертації	07.12.2020	

Студент _____ Петро КРИВЕНКО
(підпис)

Науковий керівник дисертації _____ Яна ХІЦКО
(підпис)

РЕФЕРАТ

Актуальність теми. Потребу в плануванні задач необхідно вирішувати на початковій стадії розвитку будь-якого проєкту. Проблеми недостатньої продуктивності сервера в зв'язку зі зростанням навантаження можна вирішувати шляхом нарощування потужності сервера або ж за рахунок оптимізації використовуваних алгоритмів, програмних кодів тощо. Але рано чи пізно настає момент, коли і ці заходи виявляються недостатніми та неефективними. Основними характеристиками хмарних обчислень є масштабованість, еластичність, мобільність, необмежений обсяг даних, що оброблюються та можливість нарощувати ресурси. Актуальним є аналіз існуючих підходів виділення ресурсів в хмарному середовищі та експериментальне дослідження ефективності створених правил планування задач.

Об’єкт дослідження: процес планування задач та виділення ресурсів в хмарних обчисленнях.

Предмет дослідження: методи і алгоритми планування задач та виділення ресурсів в хмарних обчисленнях.

Мета роботи: оптимізація виділення ресурсів та розподілення задач між ними, зменшення часу виконання та часу очікування обчислювальних задач.

Методи досліджень: емпіричний аналіз існуючих традиційних підходів розробки хмарних обчислень, методи аналізу і синтезу, системного аналізу.

Наукова новизна одержаних результатів полягає в наступному: запропоновано метод планування задач та виділення ресурсів в хмарних обчисленнях, на основі гібридного алгоритму, який відрізняється від відомих одночасним застосуванням алгоритмів Round Robin та Shortest Job First та дозволяє зменшити час обробки завдань на 10-17% та підвищити ефективність використання ресурсів CPU на 23%.

Практична цінність одержаних в роботі результатів дозволяють зменшити час обробки завдань при навантаженні на систему, що перевищує дозволений рівень паралельності. Це досягається шляхом застосування гібридного алгоритму динамічного виділення ресурсів та планування задач з Round Robin та Shortest Job First, який враховує час виконання та середній коефіцієнт використання ресурсів як параметри.

Апробація роботи. Основні положення і результати роботи представлені та обговорені на XIII науковій конференції магістрантів та аспірантів «Прикладна математика та комп'ютинг» ПМК-2020 (м. Київ, 18-20 листопада 2020 р.).

Структура та обсяг роботи. Магістерська дисертація складається з вступу, чотирьох розділів та висновків.

У вступі подано загальну характеристику роботи, зроблено оцінку сучасного стану проблеми, обґрунтовано актуальність напрямку досліджень, сформульовано мету і задачі досліджень, показано наукову новизну отриманих результатів і практичну цінність роботи.

У першому розділі наведено аналітичний огляд технології хмарних обчислень, їх загальні характеристики, особливості, структуру та архітектуру. Розглянуто існуючі рішення у сфері хмарних сервісів та надано огляд на ключові особливості кожного з них.

У другому розділі наведено та проаналізовано розповсюджені алгоритми та методи планування задач. Розглянуто проблему ресурсного голоду та проаналізовано алгоритми взаємного виключення для її вирішення. Детально описано запропонований алгоритм, а саме: гібридний алгоритм планування задач на основі Round Robin та Simple Job First .

У третьому розділі обгрунтовано вибір технологій реалізації хмарного ресурсу, який реалізовує запропонований метод планування задач. Описана загальна архітектура системи, детально розглянуто кожен із сервісів, що її складають.

У четвертому розділі було проведено тестування розробленого методу

планування задач на випадково згенерованих чергах задач з наперед відомим часом виконання. Отримані результати було проаналізовано та порівняно з найбільш відомими алгоритмами планування. Крім цього, проведено тестування системи хмарних ресурсів навантаженням, заміряно час роботи та кількість виділених ресурсів у системі. За результатами тестування видно, що запропонований метод планування має більшу швидкість обробки черги задач та менший час очікування в черзі.

У п’ятому розділі проаналізовано сферу хмарних обчислень. Виділено основні проблеми потенційних користувачів, сформульовано комерційне рішення, яке може ці проблеми вирішити, розроблено бізнес-модель сервісу хмарних обчислень.

У висновках проаналізовано отримані результати роботи.

Робота виконана на 96 аркушах, містить 3 додатки та посилання на список використаних літературних джерел з 23 найменувань. У роботі наведено 41 рисунок та 6 таблиць.

Ключові слова: хмарні обчислення, ефективність планування задач, алгоритми виділення ресурсів, масштабування.

ABSTRACT

Actuality of theme. The problem of planning of tasks should be solved on the early stage of the development of any project. Problems of lack of productivity of the server in connection with the growth of the installation can be tackled by increasing the power of the server, optimizing algorithms or programming code. However, sooner or later there comes a time when these measures are insufficient and ineffective. The main characteristics of cloud computing are scalability, elasticity, mobility, unlimited amount of processed data and the ability to increase resources. The analysis of the existing approaches of allocation of resources in the cloud environment and experimental research of efficiency of the created rules of planning of tasks is actual.

Object of research is the process of planning tasks and allocating resources in cloud computing..

Subject of research is methods and algorithms for resource allocation and task planning in cloud computing.

Goal of the work is to optimize the allocation of resources and the distribution of tasks between them, reducing execution time and waiting time for computational tasks.

Methods of research: empirical analysis of existing traditional approaches to the development of cloud computing, methods of analysis and synthesis, systems analysis.

Scientific novelty of the work is as follows: a method of task scheduling and resource allocation in cloud computing is proposed, based on a hybrid algorithm, which differs from the known simultaneous use of Round Robin and Shortest Job First algorithms and reduces task processing time by 10-17% and increases CPU resource efficiency by 23%.

Practical value of the results obtained in the work allows to reduce the

processing time of tasks when the load on the system exceeds the allowed level of parallelism. This is achieved by using a hybrid algorithm for dynamic resource allocation and task scheduling with use of Round Robin and Shortest Job First, which takes into account execution time and average resource utilization as parameters.

Approbation. The main provisions and results of the work were reported and discussed at the XIII scientific conference of undergraduates and graduate students "Applied Mathematics and Computing" AMC-2020.

Structure and scope of work. The master's dissertation consists of an introduction, four chapters and conclusions.

In the introduction the general characteristic of work is given, the estimation of a modern condition of a problem is made, urgency of a direction of researches is substantiated, the purpose and tasks of researches are formulated, scientific novelty of the received results and practical value of work is shown.

In the first section an analytical review of cloud computing technology, their general characteristics, features, structure and architecture is given. Existing solutions in the field of cloud services are considered and an overview of the key features of each of them is provided.

In the second section common algorithms and methods of task planning are given and analyzed. The problem of resource hunger is considered and algorithms of mutual exclusion for its solution are analyzed. The proposed algorithm is described in detail, namely: a hybrid algorithm for scheduling tasks based on Round Robin and Simple Job First.

The third section substantiates the choice of technologies for the implementation of the cloud resource, which implements the proposed method of task planning. The general architecture of the system is described, each of the services that make it up is considered in detail.

In the fourth section the developed method of task planning on randomly generated task queues with known execution time was tested. The obtained results

were analyzed and compared with the most well-known planning algorithms. In addition, the system of cloud resources was tested by load, the operating time and the number of allocated resources in the system were measured. The test results show that the proposed planning method has a higher speed of processing the queue of tasks and less waiting time in the queue.

The fifth section analyzes the sphere of cloud computing. The main problems of potential users are highlighted, a commercial solution is formulated that can solve these problems, a business model of cloud computing service is developed.

The conclusion contains brief summary of study results.

The work is performed on 96 sheets, contains 3 appendices and links to a list of used literature sources from 23 titles. The paper presents 41 figures and 6 tables.

Keywords: cloud computing, task scheduling efficiency, resource allocation algorithms, scaling.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

CPU – Central Processor Unit, центральний процесор.

API – Application Programming Interface, засоби взаємодії між окремими компонентами програмного забезпечення.

DDOS-атаки – Distributed Denial of Service, це розподілена атака на обчислювальну систему з метою спричинити її відмову.

HTTP – HyperText Transfer Protocol, це протокол прикладного рівня, де обмін повідомленнями йде за звичайною схемою «запит-відповідь».

IaaS – Infrastructure-as-a-service, це модель обслуговування, в межах якої споживачу надається можливість керувати засобами обробки та збереження, комунікаційними мережами, та іншими фундаментальними обчислювальними ресурсами.

PaaS – Platform as a Service, модель надання хмарних обчислень, при якій споживач отримує доступ до використання інформаційно-технологічних платформ.

SaaS – Software as a Service, модель поширення через хмарне середовище програм споживачам, при якій постачальник розробляє веб-програму, розміщує її й управляє нею

ЦОД – Центр обробки даних.

Віртуалізація – це технологія створення уявлення декількох комп’ютерів або серверів на базі одного фізичного комп’ютера, сервера або серверного кластера.

VM Agent – Virtual Machine Agent, агент віртуальної машини.

ВСТУП

Сьогодні хмарні технології мають все більший вплив на світ. За досить короткий проміжок часу хмарні технології стали однією із найважливіших технологій сучасності. Зараз важко здивувати когось хмарними технологіями, адже користувачів хмарних систем стає дедалі більше. Для звичайних користувачів зручність полягає у можливості зберігати дані та мати до них доступ до з будь-якого гаджету та місця. Хмарні технології дозволяють нам завантажувати, редагувати, видаляти файли у віддаленому режимі, не використовуючи ресурси наших гаджетів. Саме тому популярність хмарної інфраструктури неспинно зростає.

Для розробників програмного забезпечення хмарні обчислення привабливі можливістю зосередитися лише на прикладній функціональності, та відсутністю необхідності витрачати час та ресурси на налаштування інфраструктури та забезпечення масштабування.

При побудові хмарної інфраструктури важливо забезпечити постійний доступ до ресурсів, можливість стрімкого масштабування та ефективне розподілення навантаження на існуючих потужностях, проте існуючі рішення з розподілу ресурсів мають ряд обмежень. Проблема розподілу ресурсів повинна вирішуватись на етапі планування проєкту. Недостатньо продуктивна система може створити ряд проблем при збільшенні навантаження або навіть призвести до відмови усієї системи. Ми будемо розглядати методи планування задач та виділення ресурсів задля оптимізації роботи хмарних систем.

Метою даної роботи є аналіз, дослідження існуючих методів планування задач та динамічного виділення ресурсів у хмарних обчисленнях, а також аналіз запропонованого методу оптимізації виділення ресурсів та покращення розподілення задач між ними, оцінка переваг та недоліків даного методу, порівняння з існуючими методами виділення ресурсів.

1. АНАЛІТИЧНИЙ ОГЛЯД ТЕХНОЛОГІЇ ХМАРНИХ ОБЧИСЛЕНЬ

1.1. Технології хмарних обчислень

У наш час хмарні технології широко використовуються у різних аспектах щоденного життя. Проте, незважаючи на таку популярність, поняття хмарних технологій не є чітким та визначеним.

З технічної точки зору, хмарні технології – це спосіб організації апаратних та програмних засобів, а також набір інструментів, який надає користувачеві можливість отримувати обчислювальні потужності, для виконання поставлених задач.

Хмарні обчислення – це ресурс, що надається користувачеві у вигляді сервісу, з яким користувач може працювати віддалено. Тобто це означає, о для виконання задач користувач буде використовувати віддалені ресурси хмарної системи, а не свої локальні.

Системи хмарного обчислення будуються на основі мережевого та серверного устаткування. Це обладнання об’єднано за допомогою набору програмних комплексів, а також надає графічний інтерфейс користувачеві для керування послугами та сервісами.

У хмарній системі усі елементи працюють як єдине ціле, хоча кожна одиниця обладнання може використовуватись сама по собі. Зазвичай хмарну систему поділяють на дві частини: фронтенд та бекенд, що спілкуються між собою.

Фронтенд – це клієнтська сторона користувацького інтерфейсу до програмно-апаратної частини сервісу.

Бекенд – це «хмарна» частина системи, тобто саме програмно-апаратне рішення, де розміщені комп’ютери, сервери та системи зберігання даних, які разом об’єднані у єдину систему.

Для управління хмарною системою використовується комплекс керуючих сервісів, метою якого є керування та контроль зо навантаженням

для забезпечення стабільної роботи всієї системи. Він має набір протокол для спілкування різних елементів системи між собою.

Приклади хмарних сервісів:

1. AWS (Amazon Web Services) – це найпоширеніша в світі хмарна платформа з широкими можливостями, що надає більше 175 повнофункціональних сервісів для центрів обробки даних по всій планеті.
2. GCP (Google Cloud Platform) – хмарна патформа від Google, що пропонує безліч сервісів, таких як Google Kubernetes Engine (GKE), Google Compute Engine (GCE), Google Cloud Functions (GCF) та Google App Engine (GAE).
3. Azure – хмарна платформа компанії Microsoft. Надає можливість розробки, виконання додатків і зберігання даних на серверах, розташованих в розподілених дата-центрах.

Розглянемо детальніше кожен з сервісів.

AWS надає користувачам велику кількість різноманітних сервісів та функціональних можливостей (рис. 1) таких як: сховища і бази даних, інструменти для обчислення, штучного інтелекту та машинного навчання, за допомогою яких користувачі можуть швидко створювати різноманітні додатки, мігрувати існуючі проекти до хмари.

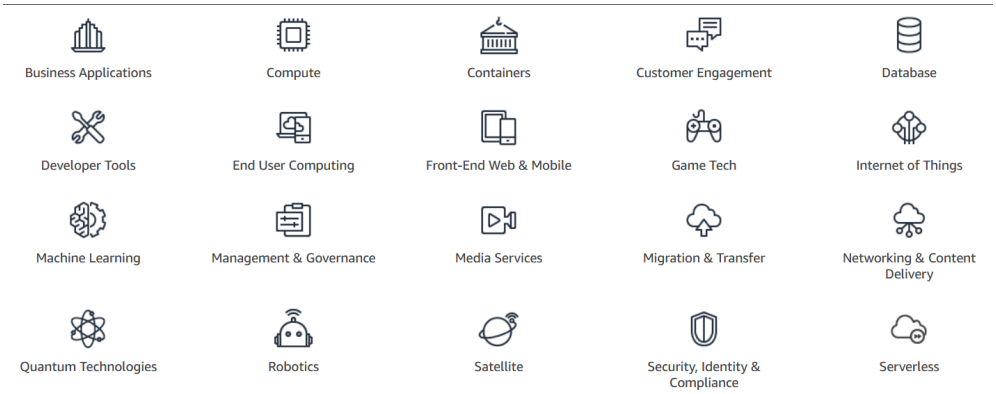


Рис. 1. Список сервісів AWS

Google Cloud Platform – це хмарні сервіси та системи від компанії Google. Надає набір хмарних сервісів таких як хмарні обчислення, зберігання даних, аналіз даних і машинне навчання та інструменти для керування ними.

Прикладом Google cloud platform сервісів є:

1. Compute Engine – сервіс, що надає можливість створювати віртуальні машини, що є максимально гнучкими у налаштуванні
2. Google Cloud Storage – надійне сховище даних, у якому можна зберігати дані, а також інтегрувати з ним різні сервіси для зберігання, використання та обробки цих даних.
3. Google Kubernetes Engine – це сервіс для запуску контейнерів, що забезпечує автоматизацію, розгортання та керування контейнерними додатками.

Azure – це хмарна платформа компанії Microsoft, що надає понад 200 хмарних сервісів, що спрямовані як на розробників програмного забезпечення так і на звичайних користувачів.

Windows Azure спочатку створювалася і розвивалася як система, що реалізує хмарну схему PaaS (платформа як сервіс), яка фактично надає замовнику лише можливість орендувати віртуалізовану інфраструктуру. Головною перевагою PaaS є орієнтованість на хмарну архітектуру, що забезпечує високу масштабованість, а також ефективне використання ресурсів. Проте Windows Azure переросла у публічну хмарну платформу і сьогодні забезпечує своїм користувачам доступність до сервісів у 83 країнах світу, в тому числі і в Україні.

1.2. Основні характеристики хмарного середовища

Хмарні технології – технології, які передбачають віддалену обробку і зберігання даних, в яких обчислювальні ресурси і потужності надаються користувачеві як інтернет-сервіс. Хмарна технологія передбачає повсюдний

і зручний мережевий доступ до спільного набору конфігурацій обчислювальних та інформаційних ресурсів (мереж передачі даних, серверів, баз даних, додатків, сервісів), які можуть бути оперативно надані та звільнені на вимогу користувача з мінімальними експлуатаційними витратами та зверненнями до провайдера [1].

Ідея хмарних обчислень з'явилася ще в 1960 році, коли Джон Маккарті висловив припущення, що коли-небудь комп'ютерні обчислення будуть проводитися за допомогою загальнодоступних утиліт. Під хмарними обчисленнями (від англ. Cloud computing) зазвичай розуміється надання користувачу комп'ютерних ресурсів і потужностей у вигляді інтернет-сервісів. Обчислювальні ресурси надаються користувачеві в «чистому» вигляді, і користувач навіть не здогадується, які саме комп'ютери обробляють його запити, під керуванням якої операційної системи це відбувається і т.д.

У хмарних обчисленнях виділяють наступні ключові характеристики [2]:

1. Самообслуговування на вимогу. Споживач самостійно вибирає, яким набором обчислювальних можливостей і ресурсів він буде користуватися (наприклад, мережеві сховища, бази даних, процесорний час, обсяг оперативної пам'яті). Також споживач може при необхідності змінювати цей набір без узгодження з провайдером в автоматичному режимі.
2. Висока еластичність (гнучкість) сервісів. Обчислювальну потужність можна легко зменшити або збільшити, виходячи з потреб користувача. У разі високого навантаження на сервіс кількість ресурсів оперативно підвищується, у разі зменшення навантаження – ресурси звільняються. Якщо певній компанії потрібно терміново збільшити обсяг обчислювальних ресурсів, то керівництву установи не доведеться витрачати кошти і час на

закупівлю і налаштування додаткового обладнання та програмного забезпечення, яке згодом може використовуватися досить рідко.

3. Можливість об’єднання ресурсів. Обчислювальні ресурси «хмарного» провайдера об’єднуються у групи з можливістю динамічного перерозподілу фізичних і віртуальних ресурсів між кінцевими споживачами. Із застосуванням сучасних технологій віртуалізації це дозволяє хмарному провайдеру легко наростити потужності і замінювати обладнання, що вийшло з ладу без зниження рівня продуктивності та надійності.
4. Облік споживання ресурсів і оплата за фактом використання. Споживачі платять тільки за фактично спожиті послуги (наприклад, за обсяг переданої інформації, пропускну здатність, обсяг даних, які зберігаються і так далі).
5. Технологічність. Можна сміливо стверджувати, що в дата-центрах постачальників хмарних послуг використовуються більш сучасні інноваційні технології, ніж у більшості звичайних дата-центрів, що надають послуги з оренди серверів, а також власних дата-центрів компаній. Ці технології дозволяють автоматично оптимізувати використання обчислювальних ресурсів і скоротити витрати на обслуговування обладнання в порівнянні з аналогічними витратами звичайних дата-центрів.
6. Високий рівень доступності. Дата-центри для хмарних обчислень, складають надійну розподілену мережу, вузли якої можуть розташовуватися в різних куточках світу, що дозволяє забезпечити максимальну доступність для користувачів із різних куточків світу.
7. Відмовостійкість. Хмарні провайдери забезпечують максимально можливий рівень відмовостійкості на різних рівнях. Кожен дата-центр має незалежне живлення та мережеве підключення. Стан кожного дата-центру знаходиться під постійним наглядом, тому у випадку певної проблеми усі запити до вузлів цього дата-центру

будуть автоматично перенапрямлені до інших дата-центрів, що у сукупності з реплікацією даних між різними дата-центрами забезпечить безперебійну роботу, навіть у випадку виходу з ладу цілого дата-центру.

Попри ряд можливостей, що надані нам хмарними технологіями, вони мають також і ряд недоліків, таких як:

1. Для роботи з хмарними сервісами потрібне постійне підключення до інтернету. Проте, в наш час це не проблема, адже зараз більш дивним виглядає людина, яка не користується інтернетом, аніж навпаки.
2. Зловмисники можуть отримати доступ до хмарної системи, використовуючи її вразливості. У будь-який момент на сервер може розгорнутись DDOS-атака від зловмисника та доступ до даних буде заблокований.

1.3. Принципи роботи

Віртуалізація – це технологія створення уявлення декількох комп’ютерів або серверів на базі одного фізичного комп’ютера, сервера або серверного кластера. Ця фізична машина називається хостом, у неї є певна конфігурація процесора, оперативної та дискової пам’яті і т.д. Фізичні ресурси за допомогою спеціалізованого ПЗ розподіляються таким чином, щоб розгорнути кілька незалежних один від одного віртуальних машин.

Гіпервізор – це процес, який відокремлює операційну систему комп’ютера і додатки від базового фізичного обладнання. Зазвичай являє собою програмне забезпечення, хоча створюються і вбудовані апаратні гіпервізори, наприклад, для мобільних пристроїв [3].

Гіпервізор є рушійною силою концепції роботи VPS і віртуалізації, дозволяючи фізичній хост-системі керувати кількома віртуальними машинами в якості гостьових ОС, що, в свою чергу, допомагає максимально

ефективно використовувати обчислювальні ресурси, такі як пам’ять, пропускна здатність мережі і кількість циклів процесора.

Переваги гіпервізорів:

1. Незважаючи на те, що віртуальні машини можуть працювати на одному і тому ж фізичному обладнанні, вони як і раніше логічно відокремлені один від одного. Це означає наступне – якщо на одній віртуальній машині сталася помилка, системний збій або шкідлива атака, то це не поширюється на інші віртуальні машини незалежно від того, встановлені вони на цьому ж комп’ютері або на інших фізичних машинах.
2. Віртуальні машини також дуже мобільні – оскільки вони не залежать від основного устаткування, їх можна переміщати або переносити між локальними або віддаленими віртуальними серверами. І зробити це набагато простіше, в порівнянні з традиційними додатками, прив’язаними до фізичного обладнання.

Існує два типи гіпервізорів: автономні гіпервізори та хостові гіпервізори. Автономні гіпервізори, запускаються безпосередньо на апаратному забезпеченні хоста для управління обладнанням і управління гостьовими віртуальними машинами. До сучасних гіпервізорів цього типу відносяться:

- Xen;
- Oracle VM Server для SPARC;
- Microsoft Hyper-V і VMware ESX / ESXi.

Хостові гіпервізори, запускаються на звичайній ОС, як і інші додатки в системі. В цьому випадку гостьова ОС виконується як процес на хості, а гіпервізор розмежовує гостьову операційну систему та операційну систему хоста [4].

Приклади гіпервізора хостового типу:

- VMware Workstation;
- VMware Player;
- VirtualBox і Parallels Desktop для Mac.

Розглянемо детальніше гіпервізор Xen (рис. 2-3), оскільки це найбільш поширений спосіб запуску віртуальних систем у хмарних сервісах.

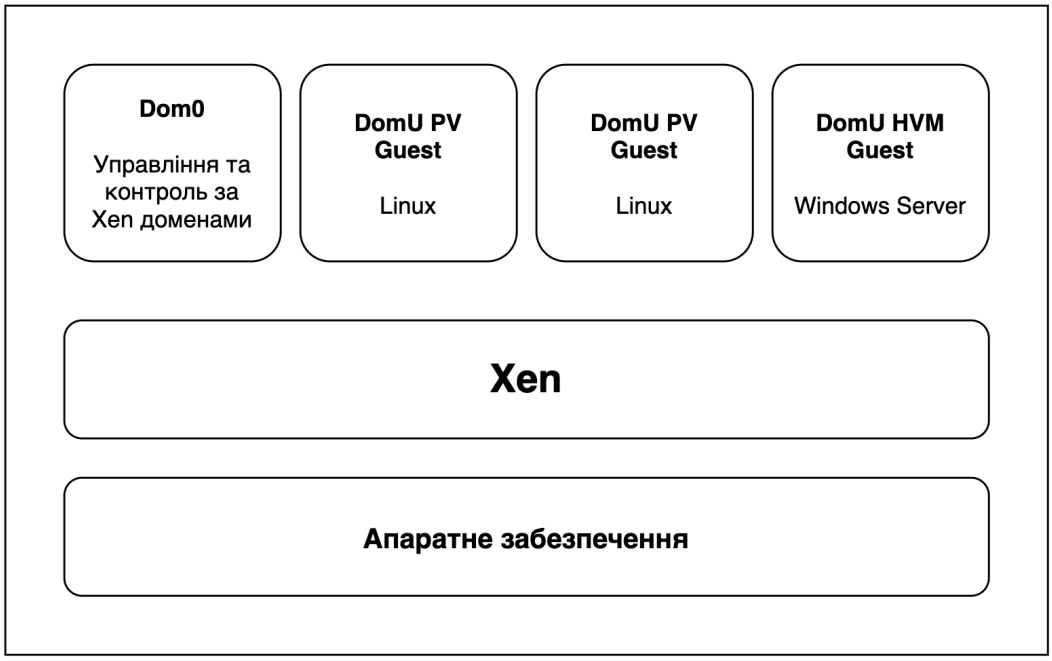


Рис. 2. Архітектура гіпервізора Xen

Xen – система паравіртуалізації, що дозволяє на одній фізичній машині (хост) запускати декілька гостьових віртуальних машин, які за допомогою Xen отримують доступ до фізичних ресурсів [5]. При цьому можливі два варіанти запуску гостьової VM – режим паравіртуалізації (PV), коли віртуальна машина запускається з модифікованим ядром і знає, що вона запущена в Xen, або віртуальна машина запускається з будь-якою модифікованою ОС, при цьому для неї запускається спеціальний процес, що емулює поведінку реального апаратного забезпечення, такий режим називається HVM.

В Xen гостьові віртуальні машини також називаються доменами. Існує один привілейований домен (Dom0), через який відбувається управління гіпервізором, робота інших гостьових машин з I/O, а також управління іншими віртуальними машинами. Гостьові віртуальні машини (DomU) можуть бути запущені тільки після завантаження Dom0.

Отже, є апаратний рівень, на якому виконується гіпервізор, поверх якого запущено кілька гостьових віртуальних машин (Dom0 & DomU). У кожної з цих віртуальних машин є доступ до CPU.

Кожна віртуальна машина працює з так званими віртуальними CPU (vCPU), це віртуалізоване уявлення одного ядра. Кількість vCPU призначається кожної віртуальної машини індивідуально або в її конфігурації, або за допомогою команди xm vcpu-set. При цьому, кількість vCPU на кожній віртуальній машині може бути не менш одного і не більше кількості ядер хоста [6].

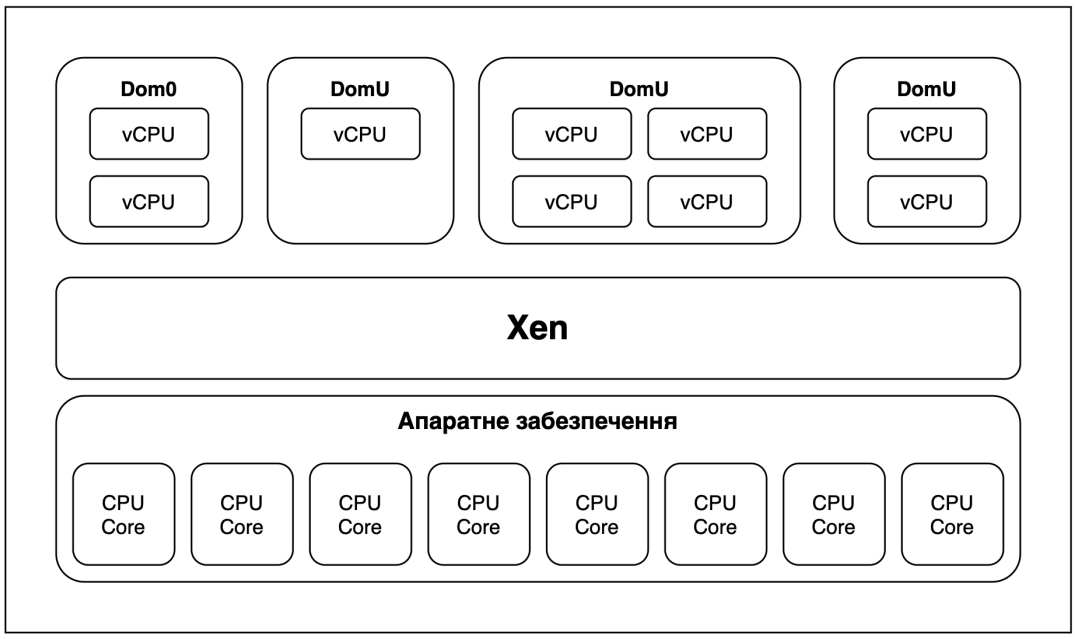


Рис. 3. Організація рівнів Xen

Завдання гіпервізора – правильно розподілити час фізичних CPU між vCPU.

Розглянемо диспетчер ресурсів в хмарних обчисленнях.

Загальна архітектура диспетчера ресурсів і його основних компонентів показана на рис. 4. Існує агент VM для кожної віртуальної машини, який визначає розподіл ресурсів на своїй віртуальній машині в кожному часовому інтервалі. Для кожного хоста є хост-агент, який отримує рішення про розподіл ресурсів всіх агентів VM і визначає остаточні розподіли, вирішуючи будь-які можливі конфлікти. Він також виявляє, коли вузол перевантажений або недовантажений, і передає цю інформацію глобальному агенту. Глобальний агент ініціює рішення про міграцію віртуальної машини, переміщаючи віртуальні машини від перевантажених або недовантажених хостів на консолідуючі хости для зменшення втрат за порушення SLA і скорочення кількості фізичних вузлів.

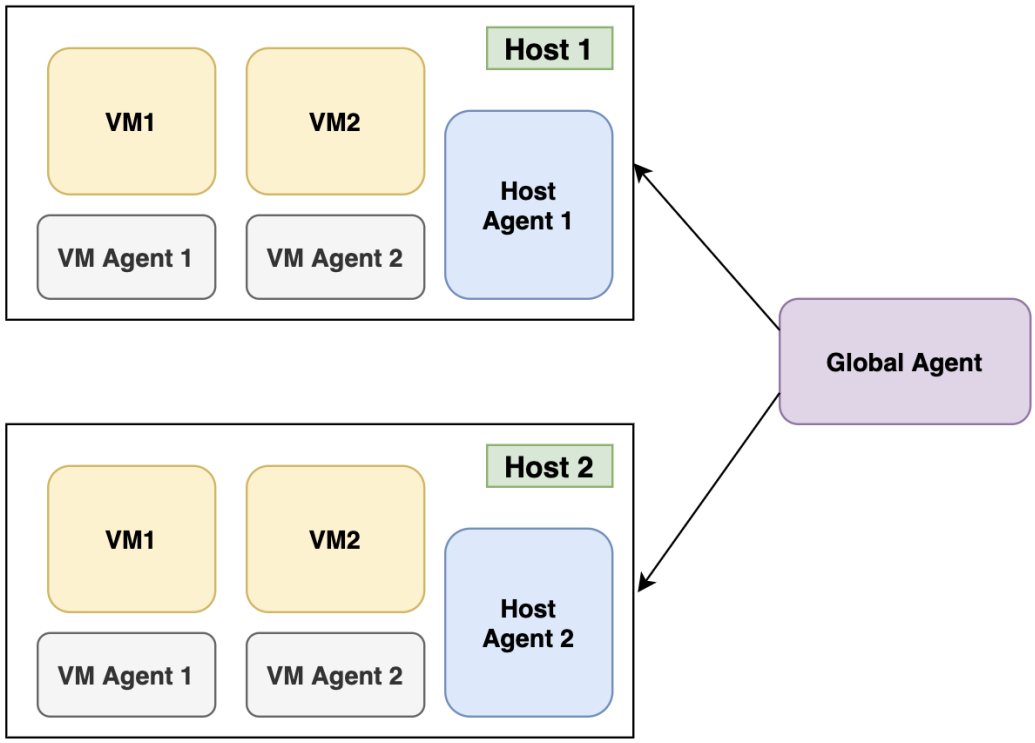


Рис. 4. Диспетчер розподілення ресурсів

VM Agent. Агент віртуальної машини відповідає за локальні рішення про розподіл ресурсів шляхом динамічного визначення загальних ресурсів, які повинні бути розподілені на його власній віртуальній машині. Рішення

про розподіл виконуються в дискретних часових інтервалах, де в кожному інтервалі визначається частка ресурсів у наступному часовому інтервалі.

Host Agent. Одним із обов'язків хост-агента – це роль арбітра. Він отримує вимоги до процесора від всіх агентів віртуальних машин, і, вирішуючи будь-які конфлікти між ними, він приймає рішення про остаточні розподіли CPU для всіх віртуальних машин. Конфлікти можуть виникати, коли вимоги CPU всіх VM перевищують загальну потужність CPU. Якщо конфлікти відсутні, остаточний розподіл CPU збігається з розподілом, що запитують агенти віртуальних машин.

Інший обов'язок хост-агента, полягає у визначенні факту перевантаженості або недовантаження хоста. Ця інформація передається глобальному агенту, який потім ініціює живу міграцію для переміщення VM від перевантажених або недовантажених хостів відповідно до алгоритму глобального розподілу.

1.4. Моделі хмарного розміщення

У наш час найбільш популярними є такі моделі хмарного розміщення [7]:

1. Public Cloud (Публічна хмара) – призначена для вільного використання широкою публікою.
2. Hybrid Cloud (Гібридна хмара) – об'єднання двох або більше хмарних інфраструктур.
3. Community Cloud (Громадська хмара) – вид інфраструктури, призначений для використання конкретним групою споживачів.

Існують три основні види хмарних послуг (рис. 5):

1. Software-as-a-service (Програмне забезпечення як послуга).
2. Platform-as-a-Service (Платформа як послуга).
3. Infrastructure-as-a-Service (Інфраструктура як послуга).

SaaS – модель продажу програмного забезпечення, при якій

постачальник розробляє веб-додаток і надає замовникам доступ до програмного забезпечення через Інтернет. Споживачеві надається можливість використання прикладного програмного забезпечення провайдера, який працює в хмарній інфраструктурі і є доступним із різних клієнтських пристроїв. Контроль і управління в цій моделі належать хмарному провайдеру.

Головними перевагами є:

- постачальник надає користувачу програмне забезпечення та програми на основі підписки;
- SaaS несе відповідальність за управління погрозами, пов'язаними з сервером, мережею і безпекою;
- використання ресурсів масштабується в залежності від потреб в послугах;
- додатки доступні майже з будь-якого пристрою, підключеного до мережі Інтернет.

Дана модель є орієнтованою на кінцевого споживача, ніж на компанію або команду розробників. Приклади:

- e-commerce системи;
- CRM системи;
- поштові сервіси.

PaaS – модель, при якій користувачеві надається комп'ютерна платформа з встановленою операційною системою і певним програмним забезпеченням. Споживачеві надається можливість використання хмарної інфраструктури для розміщення базового програмного забезпечення для подальшого розміщення на ньому нових або існуючих додатків. Головними перевагами є:

- PaaS надає платформу з інструментами для тестування, розробки та розміщення додатків у тому ж середовищі;

- PaaS зменшує обсяг коду, що необхідно розробити, автоматизує політику компанії і допомагає переносити додатки в гібридні хмари;
- з PaaS користувачі можуть краще управляти серверами, сховищами, мережами і операційними системами в цілому.

PaaS корисний для оптимізації робочих процесів. Наприклад, коли кілька розробників працюють над одним проєктом. Ця концепція особливо корисна, коли необхідно створювати індивідуальні програми. Її називають ідеальним вибором для компаній-розробників програмного забезпечення.

IaaS – це хмарне рішення, де користувачі використовують свої власні платформи та додатки в інфраструктурі постачальника послуг. Їм надається можливість використання хмарної інфраструктури для самостійного управління ПЗ. Головними перевагами є:

- користувачі оплачують IaaS за запитом;
- інфраструктура масштабується залежно від потреб у комп'ютерній потужності та пам'яті;
- постачальник послуг повинен піклуватися про віртуальні машини, включаючи всі їхні послуги, від яких він залежить.

Приклад сервісів, які використовують дані моделі послуг зображено на рис. 5.

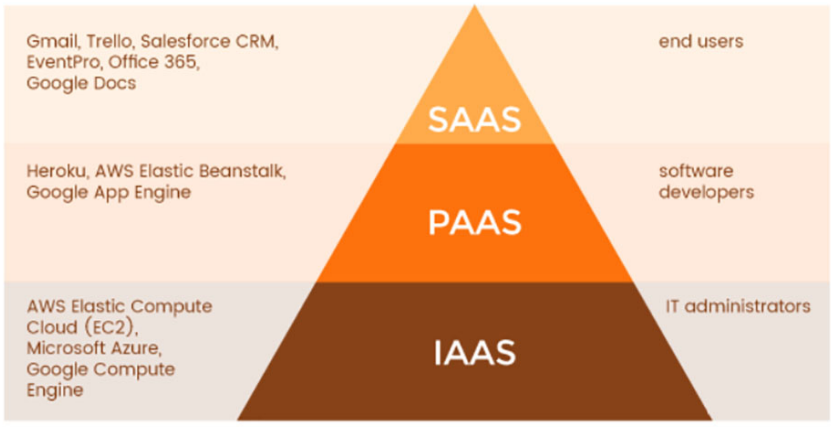


Рис. 5. Використання моделей послуг хмарними сервісами

Отже, IaaS буде найбільш вигідним для невеликих стартапів, щоб заощадити час і гроші на придбанні і створенні апаратного і програмного забезпечення. Крім того, такий варіант також буде зручним для більших компаній, у яких спостерігається динамічна завантаженість їх системи. Таким чином, кожен раз, коли необхідно отримати більше потужностей, компанія може автоматично отримати необхідну кількість ресурсів.

1.5. Переваги та недоліки

Незважаючи на очевидні переваги, концепція хмарних технологій піддається значній критиці. Основні претензії пов'язані з безпекою інформації, адже не кожному користувачу зберігання особистих даних на віддаленому сервері видається надійним. Але існує безліч переваг, які все більше переконують у потребі використанні хмарних технологій.

До ризиків використання хмарних технологій можна віднести:

1. Для роботи з хмарними сервісами потрібне постійне підключення до інтернету.
2. Користувач не завжди може налаштувати наявне програмне забезпечення під індивідуальні потреби.
3. Щоб створити власну хмарну інфраструктуру будуть потрібні дуже великі витрати, що недоцільно для нових підприємств.
4. «Хмара» – сховище даних, до яких, використовуючи вразливості системи, можуть отримати доступ зловмисники.

До головних переваг хмарних обчислень можна віднести:

1. Вся інформація доступна з будь-якого пристрою, будь то ПК, планшет, смартфон і т.д., підключеного до інтернету. Користувач не прив'язаний до певного робочого місця та гаджету.
2. Скорочення витрат на придбання дорогих потужних комп'ютерів, серверів, немає потреби оплачувати роботу IT-фахівця для обслуговування локального дата-центру.

3. Необхідні інструменти для роботи надаються автоматично веб-сервісом.
4. Високий рівень технологічності обчислювальних потужностей, який надається користувачу, дозволяє зберігати, аналізувати і обробляти дані.
5. Оплачуються сервіси тільки в міру необхідності їх використання, при цьому оплата відбувається тільки за необхідний пакет послуг.
6. Сучасні хмарні обчислення можуть забезпечувати найвищу надійність, до того ж, лише невелика кількість організацій можуть дозволити собі утримувати повноцінний дата-центр.

1.6. Масштабування в хмарному середовищі

Балансування навантаження використовується для розподілу більшого навантаження на більш дрібні вузли обробки для підвищення продуктивності системи [8]. В хмарному обчислювальному середовищі розподіл навантаження значить розподілити динамічне локальне робоче навантаження рівномірно між усіма вузлами. Балансування навантаження допомагає в справедливому розподілі обчислювальних ресурсів для досягнення високого рівня задоволеності користувачів і належного використання ресурсів. Високе використання ресурсів і правильне балансування навантаження допомагають мінімізувати споживання ресурсів. Це допомагає реалізовувати відмовостійкість, масштабованість і уникати вузьких місць. Балансування навантаження – це метод, котрий допомагає мережам та ресурсам забезпечувати максимальну пропускну здатність з мінімальним часом відгуку.

Масштабування в хмарних обчисленнях виконується на двох рівнях [9]:

- рівень віртуальної машини: створюються нові віртуальні машини на існуючих хостах для максимального використання наявних ресурсів;
- рівень хоста: горизонтальне масштабування яке відбувається за допомогою створення нових хостів, на яких будуть створюватися віртуальні машини.

Під автоматичним масштабуванням у хмарних технологіях розуміють автоматичне масштабування активних серверів у системі в залежності від навантаження на систему.

Серед цілей, для досягнення яких використовується балансування [9], потрібно виділити наступні:

- справедливість: потрібно гарантувати, щоб на обробку кожного запиту виділялися системні ресурси і не допустити виникнення ситуацій, коли один запит обробляється, а всі інші чекають своєї черги;
- ефективність використання ресурсів: всі сервери, які обробляють запити, повинні бути зайняті на 100%; бажано не допускати ситуації, коли один з серверів простоює в очікуванні запитів на обробку (в реальній практиці ця мета досягається далеко не завжди);
- скорочення часу виконання запиту: потрібно забезпечити мінімальний час між початком обробки запиту (або його постановкою в чергу на обробку) і його завершення;
- скорочення часу відгуку: потрібно мінімізувати час відповіді на запит користувача.

Ефективний алгоритм балансування повинен мати:

- передбачуваність: потрібно чітко розуміти, в яких ситуаціях і при яких навантаженнях алгоритм буде ефективним для вирішення поставлених завдань;
- рівномірне завантаження ресурсів системи;
- масштабованість: алгоритм повинен зберігати працездатність при збільшенні навантаження.

1.7. Алгоритми планування

Алгоритм планування використовується в хмарних системах для виконання черги процесів за найкоротший проміжок часу.

Кожній задачі користувача ставиться у відповідність деякий пріоритет. Чим більший пріоритет, тим вищою повинна бути реактивність завдання. Висока реактивність досягається шляхом реалізації підходу планування на основі пріоритетів (preemptive priority scheduling), суть якого полягає в тому, що за розкладом дозволяється зупиняти виконання будь-якої задачі в довільний момент часу, якщо встановлено, що інша задача повинна бути запущена негайно. Описана схема працює за таким правилом: якщо дві задачі одночасно готові до запуску, але перша має високий пріоритет, а друга — низький, то планувальник надасть перевагу першій. Друга задача буде запущена тільки після того, як завершить свою роботу перша.

Існують такі алгоритми планування задач [10]:

- First Come First Serve (FCFS);
- Round Robin (RR);
- Shortest-Job-First (SJF);
- MAX MIN та MIN MIN алгоритми.

Планування типу «перший прийшов – першим обслужений» (First-Come – First-Served, FCFS)

Таке планування є найбільш простим. Процес, що надійшов першим,

виконується до його повного завершення, при цьому використовується механізм невитискаючої багатозадачності. Невитискаюча багатозадачність – це спосіб планування процесів, при якому активний процес виконується до тих пір, поки він сам, за власною ініціативою, не віддасть керування планувальнику ОС для того, щоб той вибрав з черги інший, готовий до виконання процес.

Потім завантажується другий процес і т.д. Незважаючи на простоту, таке планування має дуже серйозний недолік: якщо виконується завдання з великим часом виконання, то інші завдання в черзі, в тому числі і системні процеси, чекають, поки воно не закінчиться, що може привести до відмов ОС. Така схема планування буде добре працювати, наприклад, в обслуговуванні систем баз даних, де короткі запити, які надходять в систему, не вимагають великого часу виконання і довжина черги не буде великою.

Алгоритм кругообігу (карусель, Round Robin, RR)

У цьому алгоритмі планування [10], відомому вже більше 50 років, процеси розташовуються в черзі, організованої як FIFO, але кожному процесу надається деякий інтервал часу, званий квантом. Після закінчення цього кванта завдання переміщується в кінець черги, а процесор починає обробляти наступну задачу. Невелика величина кванта часу сприяє поліпшенню обслуговування коротких процесів, однак сильне зменшення часу кванта призводить до зростання накладних витрат на час перемикання контекстів процесів, в той час як надлишкова величина кванта призводить до простоїв процесора через можливе закінчення роботи процесу до закінчення його кванта часу.

Найкоротша задача – спочатку (Shortest-Job-First, SJF)

У цьому алгоритмі передбачається, що система вибирає з черги ті завдання, які вимагають найкоротшого часу виконання, оскільки вони будуть вилучатися із системи за мінімальний час. Отже, при використанні моделі FIFO продуктивність системи буде підвищуватися через зменшення

середнього часу очікування завдань в черзі [10].

Хоча середній час завершення обробки черги задач, так само, як середній час очікування у цього алгоритму зазвичай нижчий, ніж у інших, він сприяє виникненню проблеми голодування для задач з великим часом обробки.

У табл. 1 зображено час роботи вищезазначених алгоритмів для прикладу з п’яти задач з різним часом виконання.

Таблиця 1

Приклад часу обробки черги задач різними алгоритмами

Задача	Оцінка часу виконання	Час завершення			Час очікування		
		FCFS	Round Robin	SJF	FCFS	Round Robin	SJF
1	50	50	150	150	0	100	100
2	40	90	140	100	50	100	60
3	30	120	120	60	90	90	30
4	20	140	90	30	120	70	10
5	10	150	50	10	140	40	0
Середнє		110	110	70	80	80	40

У другому розділі представлений гібридний алгоритм планування завдань на основі методу Shortest Job First для розділення черг на основі часу виконання завдання та вибору черги з більш швидкими завданнями для першої обробки та модифікованого методу Round Robin з динамічним квантом для обробки черг.

1.8. Висновки до розділу 1

У цьому розділі було наведено та проаналізовано принципи роботи

хмарного середовища та проведено аналіз особливостей основних моделей хмарного розміщення, а саме: програмне забезпечення як послуга (SaaS), платформа як послуга (PaaS) та інфраструктура як послуга (IaaS). Виділено переваги та недоліки хмарних технологій, які показують, що завдяки об’єднанню ресурсів та непостійному характеру споживання з боку користувачів, можна забезпечити економію в масштабі системи, використовуючи менші апаратні ресурси, ніж при виділенні апаратних потужностей для кожного споживача, що є дуже ефективним, а завдяки автоматизації модифікації виділення ресурсів значно знижуються витрати на обслуговування.

Наведено короткий опис основних алгоритмів планування задач, а також підкреслено їх основні недоліки та переваги.

2. ФОРМУЛЮВАННЯ МОДИФІКАЦІЇ МЕТОДУ ПЛАНУВАННЯ ТА ДИНАМІЧНОГО ВИДІЛЕННЯ РЕСУРСІВ

2.1. Проблема ресурсного голоду

Голодування ресурсу є проблемою, що зустрічається у хмарних обчисленнях, коли певний набір задач може ніколи не бути переданим до виконання. Ресурсний голод може бути викликано помилками в алгоритмах планування або взаємного виключення. Ще однією причиною ресурсного голоду є витік ресурсів, який може бути спровоковано навмисно за допомогою атаки типу відмова в обслуговуванні, такою як fork-бомба (шкідлива або помилково написана програма, яка нескінченно створює свої копії системним викликом fork, які зазвичай також починають створювати свої копії).

Проте основною причиною ресурсного голоду є примітивні алгоритми планування задач. Наприклад, якщо погано спроєктована багатозадачна система завжди перемикається між першими двома завданнями, то у такому випадку третій задачі не вистачає процесорного часу. Алгоритм планування, який є важливою частиною хмарних сервісів, повинен розподіляти ресурси рівномірно; тобто алгоритм повинен розподіляти ресурси так, щоб усім задачам завжди вистачало необхідних ресурсів для їх виконання.

Також голод можливий у високо-навантажених пріоритетних системах, у яких виконується безліч дрібних задач. Якщо під час виконання поточного завдання з’являється більш пріоритетне завдання, то поточне завдання буде перервано з ціллю надати ресурс для обчислення більш пріоритетної задачі. Це створює зайві накладні витрати через необхідність провести додаткові перемикання контексту. Планувальник також повинен вміщувати кожну вхідну задачу у певне місце в черзі, що також створює додаткові накладні витрати.

Велика кількість планувальників задач у хмарних системах використовують концепцію пріоритетів задач, а отже ці алгоритми є потенційно схильними до проблеми ресурсного голодування. Задача А, що має високий пріоритет буде виконуватися перед задачею В з низьким пріоритетом. Якщо у такому випадку задача з високим пріоритетом блокується і ніколи не завершується, задачу з низьким пріоритетом ніколи не буде завершено – вона буде страждати від ресурсного голоду. Якщо при цьому існує задача С з ще більш високим пріоритетом, який залежить від результату задачі В, то ця задача С може ніколи не завершитися, навіть якщо це найважливіший процес в системі. Цей стан називається інверсією пріоритету.

Можливим вирішенням проблеми голодування для алгоритмів планування за пріоритетами є використання методу старіння. Старіння – це метод поступового підвищення пріоритету процесів, які довгий час чекають в системі.

Сучасні алгоритми планування зазвичай містять код, який гарантує, що всі задачі отримають мінімальну кількість кожного важливого ресурсного часу, щоб запобігти виснаженню будь-якої задачі.

Виснаження зазвичай викликається тупиковою ситуацією, яка призводить до зависання задач. Це стається коли дві або більше задачі зайшли в глухий кут, коли кожна з них не може бути виконаною через очікування на іншу задачу з того ж набору.

Для вирішення таких тупикових ситуацій використовуються алгоритми взаємного виключення. Проблема, яку вирішують взаємні виключення, – це проблема спільного використання ресурсів, тобто управління доступом декількох задач до спільного ресурсу, коли кожна з задач потребує виняткового контролю над цим ресурсом при виконанні своєї роботи. Найбільш поширене рішення цієї проблеми – це робити загальний ресурс доступним тільки у тому випадку, коли задача знаходиться в певному сегменті коду, званому критичної секцією, і контролювати доступ

до загального ресурсу, контролюючи кожне взаємне виконання тієї частини задачі, в якій буде використовуватися ресурс.

Отже, проблема ресурсного голодування алгоритмів планування задач часто зустрічається у алгоритмів з пріоритетними чергами, а найменш розповсюджена проблема голодування у алгоритмів FIFO, адже голод не може існувати, якщо немає пріоритетної черги. Порядок розподілення одиниць основується на часі прибуття процесу.

2.2. Прогнозування часу необхідного на обробку задачі

Отримання оцінок часу виконання алгоритму є однією з найважливіших задач ефективного розпаралелювання і оптимального розподілу потоків управління в багатопроцесорних конфігураціях. Традиційно саме час виконання є основним, хоча і не єдиним, індексом продуктивності.

Будь-яка комп’ютерна програма описує деяке сімейство алгоритмів, вибір яких визначається спрацьовуванням умовних операторів. Випадок відсутності умовних операторів означає реалізацію одного алгоритму або однієї послідовності операцій.

Спрацювання умовних операторів визначається тільки вхідними даними і результатами їх обробки, що, в кінцевому рахунку, робить таку залежність від вхідної інформації винятковою. При цьому результат на виході програми буде інваріантний до вибору комп’ютера, на якому буде виконуватися програма, за умови незмінності вхідних даних.

На теоретичну оцінку часу реалізації алгоритму впливають наступні фактори:

1. Часткова невідповідність структур і конструкцій мови програмування, обраного для реалізації алгоритму, особливо в частині структур і типів даних, і реальної системи команд

процесора і підтримуваних ним типів даних – проблема, відома в області архітектур процесорів під назвою семантичний розрив.

2. Наявність архітектурних особливостей, які істотно впливають на спостережуваний час виконання програми, таких як стекова обробка, конвеєр команд і конвеєр даних, наявність декількох рівнів швидких буферів пам’яті (кеш-пам’ять), апаратні і програмні кошти передвиборки команд і даних і т.д.
3. Відмінність в часі виконання різних машинних команд, обумовлених різною складністю внутрішніх алгоритмів реалізації апаратних обчислень. Слід відзначити, що ця проблема частково знімається для сучасних RISC процесорів, в яких більшість машинних команд виконується за фіксовану кількість тактів.
4. Відмінність часу реального виконання однієї команди в залежності від типів даних.
5. Відмінність у часі виконання однієї команди, в залежності від значень операндів.
6. Неоднозначність компіляції вихідного тексту, зумовлена як специфікою обраного компілятора, наприклад, в частині підтримки їм різних методів оптимізації коду, так і особливостями його налаштування.
7. Додаткові часові затримки, зумовлені обраної середовищем реалізації, наприклад затримки, пов’язані з квантуванням часу завдань і т.д.

Перераховані фактори суттєво ускладнюють теоретичне побудування функції часу виконання, проте, спроби різного підходу до їх урахування призвели до появи різноманітних методів побудови часових оцінок.

В методах оцінки складності задач та часу їх виконання використовується поняття трудомісткості алгоритмів $F_a(N)$ – кількість елементарних операцій, що здійснюється алгоритмом для вирішення певної проблеми в заданій формальній системі [11].

В якості «елементарних» операцій пропонується використовувати наступні:

1. Просте присвоювання $a \leftarrow b$.
2. Одновимірна індексація $a[i]$: (*адреса* (a) + *i*довжина елементу*).
3. Арифметичні операції (*, +, –, /).
4. Операції порівняння $a < b$.
5. Логічні операції {*or*, *and*, *not*}.

Спираючись на ідеї структурного програмування, виключимо команду переходу за адресою, вважаючи її пов’язаною з операцією порівняння в конструкції розгалуження.

Після введення елементарних операцій аналіз трудомісткості основних алгоритмічних конструкцій у загальному вигляді зводиться до наступних положень:

1. Конструкція «слідування».

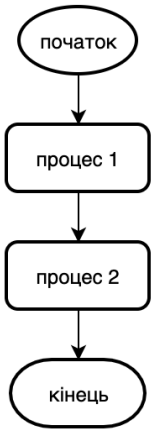


Рис. 6. Блок-схема конструкції «слідування»

Трудомісткість конструкції визначається як сума трудомісткостей блоків, що слідують один за одним:

$$F = \sum_{k} f_k , \tag{1}$$

де k – кількість блоків.

2. Конструкція «розгалуження».

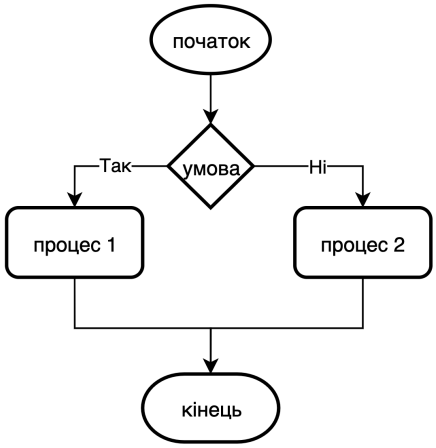


Рис. 7. Блок-схема конструкції «розгалуження»

Загальна трудомісткість конструкції «розгалуження» вимагає аналізу ймовірності виконання переходів на блоки «true» і «false» і визначається як:

$$F = f_{true} \times p + f_{false} \times (1 - p), \quad (2)$$

де p – ймовірність виконання умови та переходу на гілку *true*, трудомісткість якої становить f_{true}; f_{false} – трудомісткість гілки false.

3. Конструкція «цикл».

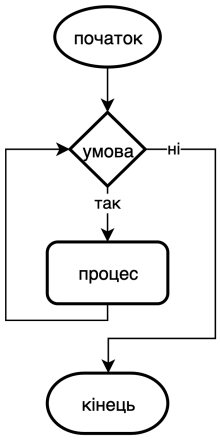


Рис. 8. Блок-схема конструкції «цикл»

Після зведення конструкції до елементарних операцій її

трудомісткість визначається як:

$$f = 1 + 3N + N \times f_{body}, \quad (3)$$

де N – кількість ітерацій, f_{body} – трудомісткість тіла циклу.

Порівняння двох алгоритмів по їх функції трудомісткості вносить деяку помилку в результати. Основною причиною цієї помилки є різна частота зустрічальності елементарних операцій, що породжується різними алгоритмами і відмінність в часах виконання елементарних операцій на реальному процесорі. Таким чином, виникає задача переходу від функції трудомісткості до оцінки часу роботи алгоритму на конкретному процесорі: за заданою функцією трудомісткості алгоритму $F_a(D_A)$ знайти час роботи програмної реалізації алгоритму $T_A(D_A)$, де D_A – множина задач, задана в формальної системі.

Для вирішення цієї задачи використовуються наступні методи [12].

Метод типових задач (Гібсона)

Це один з перших методів, який представляє собою спробу отримати універсальний підхід до отримання часових оцінок, що враховує тільки тип розв'язуваної задачі.

Ідея методу полягає в тому, що в рамках фіксованого типу завдань, наприклад завдань, пов'язаних з науково-технічними розрахунками, середній час на одну узагальнену операцію буде стійким через використання однакових типів даних.

Метод передбачає проведення сукупного аналізу досліджуваного алгоритму по трудомісткості і перехід до часової оцінки його програмних реалізацій на основі приналежності розв'язуваної задачі до одного з наступних типів:

- завдання науково-технічного характеру з переважанням операцій з операндами дійсного типу;
- завдання дискретної математики з оволодінням операцій з операндами цілого типу;

- завдання баз даних з переважаючими операціями з операндами строкового типу.

Далі на основі аналізу множини реальних програм для вирішення відповідних типів завдань визначається частота зустрічальності операцій (рис. 9). Отримані результати лежать в основі створення відповідних тестових програм, що породжують задану частоту зустрічальності операцій для заданих типів даних.

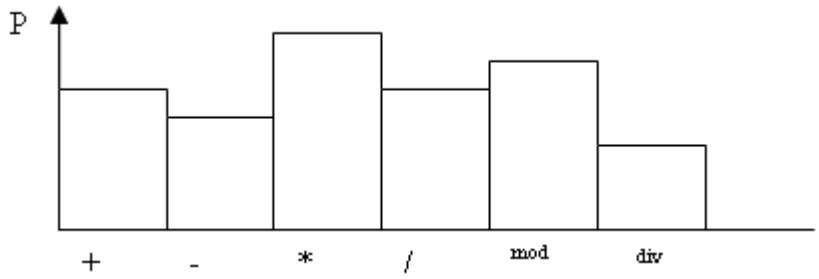


Рис. 9. Можливий вигляд частоти зустрічальності операцій

На основі експериментів з цими тестовими програмами і виділяється середній час на узагальнену операцію в даному типі завдань, далі оцінюється загальний час роботи програмної реалізації алгоритму для даного комп'ютера і цього середовища за формулою:

$$T_A(N) = F_a(N) \times \bar{t}_{op}, \qquad (4)$$

де $\bar{t}_{op}$ – середній час виконання операції.

Перевага методу полягає в тому, що значення часу визначається одноразово для даної мови, компілятора, ОС і комп'ютера, і потім використовується для отримання часових оцінок програмних реалізацій алгоритмів на основі приналежності розв'язуваної задачі до одного із зазначених типів.

Такі оцінки, очевидно, не володіють великою точністю, але можуть бути досить просто отримані для будь-якої програмної реалізації виконуваної на даному комп'ютері.

Метод пооперaційного аналізу

Ідея методу пооперaційного аналізу полягає в поданні функції трудомісткості алгоритму у вигляді суми трудомісткості з базових операцій і визначення середнього часу виконання кожної з них в вибраному середовищі реалізації. Метод пооперaційного аналізу включає в себе наступні етапи:

1. Отримання функцій трудомісткості для кожної з використовуваних алгоритмом базових операцій з урахуванням типів даних. Отримання таких післяопераційних функцій вимагає певних витрат, проте, в деяких випадках такий підхід дозволяє отримати значущі результати щодо вибору раціональних алгоритмів.
2. Експериментальне визначення середнього часу виконання даної базової операції на конкретній обчислювальній машині в середовищі обраної мови програмування і операційної системи за допомогою спеціальної тестової програми.
3. Очікуваний час виконання програмної реалізації розраховується як сума добутків пооперaційної трудомісткості на середній час операцій:

$$T_A(N) = \sum_i F_{ai}(N) \times \overline{t}_i . \qquad (5)$$

Помилки прогнозу в даному методі, незважаючи на його достатню детальність, пов’язані з тим, що реальний потік операцій, породжуваних алгоритмом, не завжди збігається з потоком в експериментальній програмі, яка визначає середній час виконання базової операції.

Такі розбіжності можуть обумовлені, наприклад, конвеєрною архітектурою процесора, наявністю блоку попередньої вибірки команд, внутрішніми алгоритмами управління кеш-пам’яттю і іншими особливостями архітектури.

Експериментальний метод отримання часових оцінок на основі функції трудомісткості

На основі функції трудомісткості алгоритму [4] і ряду експериментів з його програмною реалізацією можливе отримання часових оцінок на основі середнього часу виконання узагальненої базової операції. Метод детально розроблений, і являє собою сукупність наступних етапів:

1. Сукупний аналіз трудомісткості алгоритму без поділу на операції; на цьому етапі визначається трудомісткість $F_a(N)$ в узагальнених базових операціях прийнятої моделі обчислень.
2. Проведення обчислювальних експериментів з програмною реалізацією алгоритму; на цьому етапі для кожного з обраних значень розмірності задачі N_{exp}, проводиться певна кількість експериментів, з різними вихідними даними, в ході яких визначаються часи виконання, на основі яких розраховується середній експериментальний час виконання T_{exp}.
3. Розрахунок середнього часу; в рамках цього етапу на основі відомої функції трудомісткості алгоритму в середньому розраховується середній час на узагальнену базову операцію $\bar{t}_a$, що породжується даним алгоритмом, компілятором, операційним середовищем і комп'ютером для даної розмірності, і по всьому експерименту в цілому.
4. Прогноз часової ефективності; на цьому етапі, припускаючи, що середнього часу виконання узагальненої базової операці є стійким, отримані результати можна інтерполювати або екстрапольовані на інші значення розмірності задачі за формулами:

$$\overline{t_a} = \frac{T_{\exp}(N_{\exp})}{F_a(N_{\exp})}, \tag{6}$$

$$T_A(N) = \overline{t_a} \times F_a(N). \tag{7}$$

Таким чином, представлені методики прогнозування часу виконання

алгоритмів дозволяють оцінити час виконання, базуючись на ймовірнісній оцінці результатів логічних операцій, які залежать виключно від вхідних даних.

2.3. Гібридний алгоритм планування задач

В даному розділі розглянемо запропонований алгоритм планування задач, що базується на алгоритмі Round Robin з динамічним квантом часу та Simple Job First.

2.3.1. Модифікований Round Robin з динамічним значенням кванту часу

Квант часу – числове значення, що визначає, як довго може виконуватися процес до того моменту, поки він не буде витіснений.

Занадто велике значення кванта часу призведе до погіршення інтерактивної продуктивності системи, а занадто мале значення кванта часу призведе до зростання накладних витрат на переключення між процесами.

Round Robin – це алгоритм, який займається розподілом навантаження розподіленої обчислювальної системи, що використовує впорядкування її елементів по круговому циклу.

Даний алгоритм є одним з важливих алгоритмів планування в плануванні завдань. Round Robin – превентивний алгоритм планування, що використовує квантум часу для виконання процесу. Цей алгоритм є найпопулярнішим у наш час з алгоритмів планування, має надзвичайно мале ресурсне голодування. Розглянемо його переваги [13]:

1. Фактична можливість здійснення в системі, оскільки немає залежності від часу пакета.
2. Відсутня проблема можливого голодування.
3. Всі роботи отримують розподіл ресурсів процесора.

Хоч цей алгоритм і є найпопулярнішим, він має ряд недоліків саме з квантом часу:

1. Чим вище квант часу, тим вище час відгуку в системі.
2. Чим менше квант часу, тим вище витрати перемикання контексту в системі.
3. Вибір ідеального кванта часу – дійсно дуже складне завдання в системі.

Планування за принципом RR припускає диспетчеризацію процесів за принципом FIFO, але кожен процес отримує часовий квант, протягом якого він може використовувати ресурси ЦП. Якщо завершення процесу не відбувається після закінчення кванта часу, то цей процес переводиться в кінець списку готових до виконання процесів, а ресурси ЦП надаються наступного процесу зі списку. Такий алгоритм планування підходить, наприклад, для роботи з поділом часу, коли система повинна гарантувати прийнятні часи відповіді для всіх інтерактивних користувачів.

Очевидно, що для даного алгоритму планування основне питання полягає у визначенні розміру кванта часу, і чи слід робити його фіксованим або змінним. Очевидно, якщо квант часу вибирається занадто великим, то система RR фактично вироджується в FIFO, тому що кожному процесу виділяється достатньо часу для завершення. Якщо ж квант часу вибирається занадто малим, то контекстні перемикання починають відігравати домінуючу роль, що в підсумку погіршує характеристики системи.

Задля оптимізації даного алгоритму пропонується виконати найважчу задачу для цього алгоритму – підібрати досконалий квант часу, методом перетворення його у динамічний.

Динамічне значення кванту часу може вирішити проблему неоптимальної зміни контексту, також, формування пріоритетної черги з задачами, що вимагають найменше процесорного часу збільшить пропускну здатність. Ймовірно, такий підхід до використання квантуму часу у алгоритмі Round Robin зменшить час очікування, час відповіді та пропускну

здатність, а також зменшить ресурсне голодування.

Розглянемо динамічний квантум часу циклічного алгоритму Round Robin:

1. У стандартних реалізаціях алгоритму Round Robin використовується константне значення квантуму часу, частіше за все 10-100 мс, що може призвести до втрати продуктивності виконання задач. У випадку замалого значення виникатиме черезмірне перемикання контекстів, що знизить загальну продуктивність. Водночас у випадку завеликого значення пропускна спроможність знижується та зростає час відповіді, що наближує алгоритм за цими параметрами до iFCFS.
2. Квантум часу з кожною ітерацією адаптується до часу необхідного для виконання задачі на CPU (burst time) на основі задач знайдених у черзі.

Отже, очевидно, що використання динамічного квантуму часу у алгоритмі Round Robin є досить доцільним. Використовуючи такий підхід, можна оптимізувати одразу декілька напрямів, таких як:

- ресурсне голодування;
- час очікування та відповіді;
- пропускна здатність.

2.3.2. Гібридний алгоритм Simple Job First та Round Robin

Запропонований алгоритм виділення ресурсів є гібридним серед двох алгоритмів Simple Job First та Round Robin.

Він дозволяє використати переваги обох алгоритмів та зменшити їх недоліки. При поєднанні цих алгоритмів, середній час обробки задач буде менший, ніж у стандартному Round Robin, а голодування ресурсів буде меншим, ніж у стандартному Simple Job First.

Для реалізації планування за допомогою Round Robin ми працюємо з чергою задач як із FIFO. Планувальник обирає перший процес з готової черги і встановлює таймер переривання через час, що дорівнює одному

кванту та відправляє процес на обробку. Проте задача може завершитись раніше виділеного їй часу, тоді планувальник перейде до наступного процесу у черзі. У випадку, якщо задачі необхідно більше часу, ніж було виділено, тоді буде виконано зміну контексту та переведено задачу до хвоста цієї черги, після чого планувальник перейде до наступної задачі, що буде першою у черзі [13].

Фіксований квант часу призводить до втрати продуктивності виконання задач. У випадку замалого значення виникатиме надлишкове перемикання контекстів, що знизить загальну продуктивність, водночас у випадку завеликого значення пропускна спроможність знижується та зростає час відповіді [14].

Гібридний алгоритм полягає у виділенні пріоритетної черги за значенням часу, необхідного на завершення кожної із задач у цій черзі. Можливість застосувати різні кванти часу для алгоритму Round Robin дає можливість повністю обробити пріорітетну чергу за два проходи, а потім частково опрацювати усі можливі задачі менш пріоритетної черги [10,13].

Алгоритм формулюється наступним чином. Нехай $Q = [t_1, t_2, \dots, t_n]$ – вхідна черга задач, де кожна задача характеризується часом *t*, необхідним на її завершення. Для оброблення даної черги необхідно виконати наступні кроки:

1. Відсортувати чергу задач Q за зростанням часу обороту, тобто

$$t_1 < t_2 < \dots < t_{n-1} < t_n . \tag{8}$$

2. Знайти медіану t_M черги задач.
3. Знайти початковий квант часу q_1 – час задачі з максимальним часом виконання, що є меншою за значення $\frac{t_M}{2}$.
4. Розділити чергу *Q* на дві нові черги так, щоб

$$Q_1 = [t_1, t_2, \dots, t_i], \text{ де } t_i \le 2q_1; \tag{9}$$

$$Q_2 = [t_{i+1}, \dots, t_n], \text{ де } t_{i+1} > 2q_1 . \tag{10}$$

5. Виконати перший прохід чергою Q1 за допомогою алгоритму RR з квантом часу $q = q_1$.
6. Виконати другий прохід чергою Q_1 алгоритмом RR з квантом часу $q = \max(Q_1)$.
7. Виконати прохід чергою Q_2 алгоритмом RR з квантом часу $q = t_M$.
8. Створити нову чергу задач $Q = Q2 + N$, де N – черга з новими задачами, що були отримані протягом обробки поточної ітерації.
9. Повторювати алгоритм циклічно починаючи з п.1 доки не отримаємо порожню чергу Q.

На рис. 10 наведено блок-схему запропонованого алгоритму.

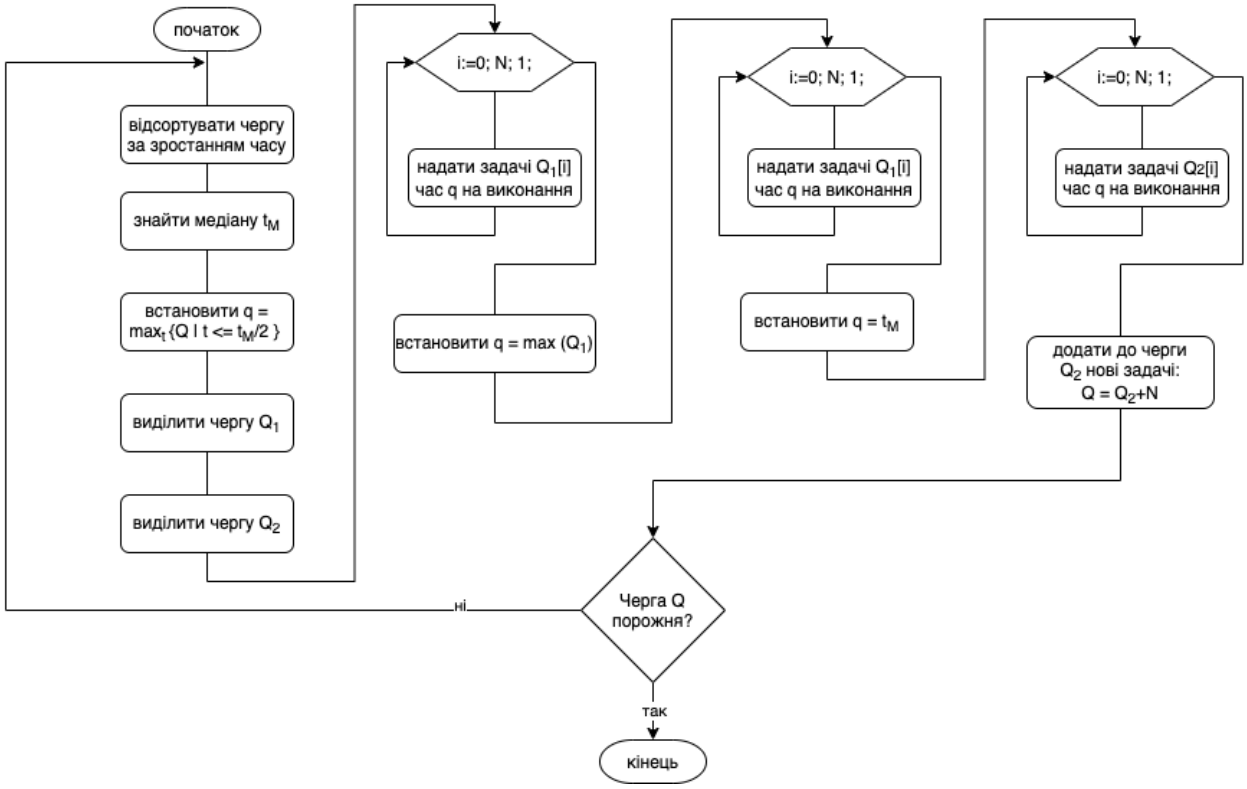


Рис. 10. Блок-схема гібридного алгоритму обробки задач

Цей алгоритм надає пріоритету найпростішим задачам, повністю виконуючи їх у рамках ітерації алгоритму, а також частково оброблює більш ресурсоємні задачі, завдяки чому не допускається ефект голодування ресурсів.

2.4. Алгоритми та методи моніторингу та виділення нових ресурсів

Оскільки створенням бази даних, машин, серверів додатків в хмарі займаються тепер самі користувачі, то є ризик отримати безперервне і слабо контрольоване розростання числа цих об’єктів, а те, що вони автоматично (відповідно до політики) можуть мігрувати на інші машини, зупинятися і відновлювати роботу, – ще більше ускладнює управління і контроль.

Не варто забувати і про еластичність, зворотною стороною якої є те, що розміри віртуальних машин і кластерів бази даних можуть динамічно збільшуватися або зменшуватися. Необхідно також контролювати використання дискового простору, оперативної пам’яті, процесорів в хмарі і заздалегідь передбачити вичерпання цих ресурсів.

Під хмарою розуміється стандартизація створюваних об’єктів на основі механізму шаблонів або збірок, але, почавши жити своїм життям, далі ці об’єкти змінюють свої характеристики, що теж треба відстежувати. Крім того, не слід забувати, що все це величезна кількість баз, машин, серверів додатків, яку треба періодично оновлювати.

При роботі як самих об’єктів, так і процедур їх створення можуть виникати помилки, з якими повинні розбиратися адміністратори хмари. Нікуди не зникають традиційні проблеми адміністрування віртуальних машин, баз даних, серверів додатків, самих додатків і т. д. – від того, що база даних створена в хмарі, обсяг роботи з її налаштування, діагностування, обслуговування не зменшується.

Для моніторингу та управління хмарою [15] і його елементами потрібні інструменти, що дозволяють управляти його об’єктами, причому управління і моніторинг повинні забезпечуватися на рівні груп об’єктів, так як працювати індивідуально з такою кількістю об’єктів неможливо.

Для управління всіма етапами цього життєвого циклу призначене рішення Oracle Enterprise Manager 12c, за допомогою якого за три наступні

кроки створюється хмара:

- планування і створення хмарної інфраструктури;
- створення і каталогізація в бібліотеці ПЗ шаблонів, збірок і процедур розгортання, створення користувачів;
- моніторинг і управління хмарою, тарифікація і білінг.

Для створення процедур розгортання бази даних: одиночних, кластерних або з автоматичним управлінням системою зберігання (ASM, Automatic Storage Manager) – OEM (Oracle Enterprise Manager) запускає Database Configuration Assistant, в якому адміністратор описує всі параметри майбутньої бази і налаштування СУБД.

Для побудови шаблонів віртуальних машин OEM використовує Oracle VM Template Builder, який створює шаблон на основі існуючої фізичної або віртуальної машини.

Для формування розподілених додатків в хмарі створюються збірки (Assembly), що описують всі віртуальні машини такого додатка і правила їх взаємодії (імена, конфігурації мережі, конфігурації дисків і т. д.). Це робиться за допомогою програми Oracle Virtual Assembly Builder, що дозволяє описати всі компоненти такого додатка і зв’язку між ними, після чого ця програма генерує набір шаблонів і даних, об’єднаних в збірку.

Перед тим, як опублікувати об’єкти в бібліотеці ПЗ, їх треба протестувати, і для цього пропонується набір засобів функціонального і навантажувального тестування як бази даних, так і всієї програми. Відповідальність за якість пропонованих кінцевому користувачеві сервісів лежить на адміністраторі самообслуговування. Після наповнення бібліотеки ПЗ і опису ролей і квот користувачів хмара готова до роботи.

Хмара ділиться на зони. Кожна зона являє собою в певному місці географічно розташований дата-центр. Крім того, у кожної зони є свій Storage (Сховище). Зони діляться на поди.

Под передбачає об’єднання групи кластерів всередині однієї мережі [9]. Так само поділ на поди дає можливість включення / відключення

відразу групи кластерів.

Кластер складається з ідентичних хостів, які мають однакову версію ОС і використовують загальний гіпервізор, і має свій Primary Storage, який використовується віртуальними машинами. Достатня кількість хостів в кластері надає можливість зробити високу доступність, а так же розподіл навантаження.

Storage використовується для зберігання шаблонів віртуальних машин, ISO образів, а також знімків. Сучасні хмари підтримують багато сучасних гіпервізорів, такі як XenServer, KVM Server, Oracle VM і VMware. Хмара підтримує моніторинг використовуваних ресурсів і при їх вичерпанні посилає повідомлення адміністратору. Так само при великому навантаженні на певні компоненти можливий перерозподіл навантаження.

Для порівняння алгоритмів управління ресурсами в cloud системах розроблена спеціалізована програмна середа CloudSimulation. У конфігураціонному файлі створюється структура хмари (рис. 11), у файл evm_templates задаються шаблони, на основі яких будуть створюватися запити на старт віртуальних машин. CloudSimulation реалізований за принципом, аналогічному CloudStack (Хмарний сервіс). У файлі конфігурації задаються зони, поди, кластери, і так само хости. Для кластера задається розмір Primary Storage. Для хоста – кількість CPU, CPU speed і розмір RAM.

Як час емуляції обрані секунди. В файлі конфігурації шаблонів віртуальних машин описуються необхідні параметри віртуальної машини:

- кількість CPU;
- швидкість CPU;
- розмір RAM;
- розмір жорсткого диска;
- ідеальний час роботи, необхідний для виконання завдання віртуальної машини;

- час в секундах між двома запусками віртуальних машин за цим шаблоном.

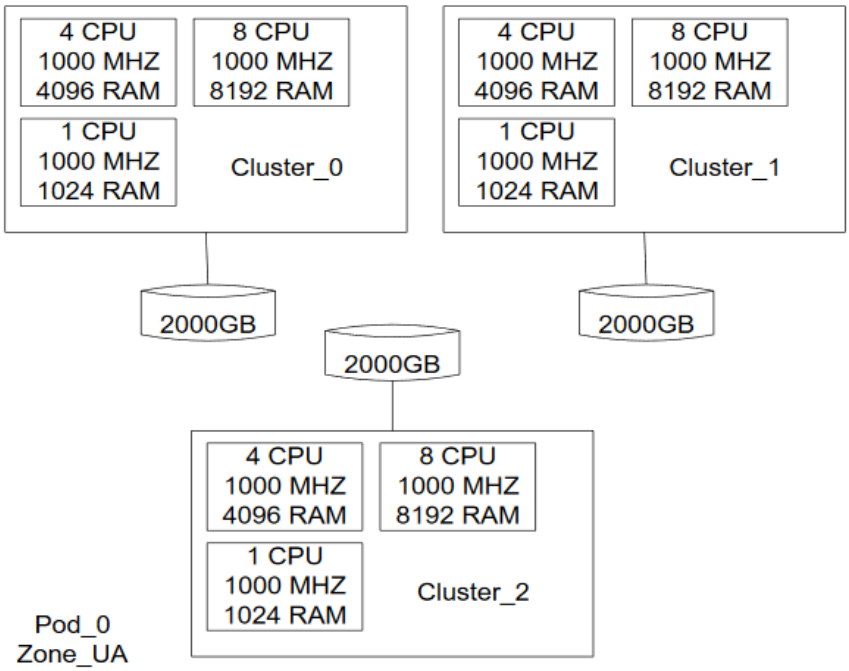


Рис. 11. Схема тестованої хмари

При моделюванні тестувалося 3 різних алгоритму пошуку ресурсу:

- random – випадковий вибір ресурсу для завдання;
- LF (least fit) – вибирається найменш підходящий ресурс з відповідних;
- FF (first fit, best fit) – вибирається найбільш підходящий ресурс;
- при часі моделювання 20 000 сек: На момент 20000 всього було створено 337 заявок на запуск VM.

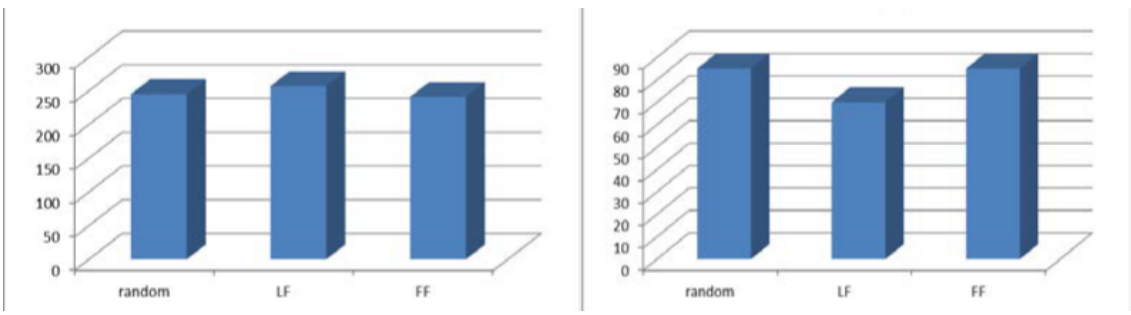


Рис. 12. Кількість виконаних VM та кількість запущених VM

При часі моделювання 10 000 сек.: На момент 10000 всього було створено 170 заявок на запуск VM.

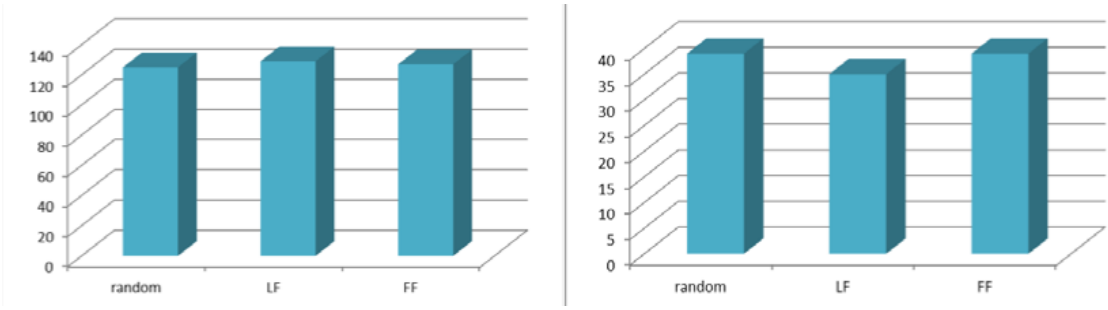


Рис. 13. Кількість виконаних VM та кількість запущених VM

Як видно з рис. 12 і 13 алгоритм LF показав найбільший коефіцієнт виконання VM при різних часових рамках. Це пояснюється тим, що маленькі (не вимогливі) заявки потрапляють на вже завантажені ресурси, а вільні ресурси можуть виконати найвибагливіші заявки. Алгоритм LF виділяється тим, що максимально навантажуються вже використовувані ресурси, тобто частина ресурсів буде працювати на повну потужність, тоді як інші будуть простоювати і можуть включати режими зниженого енергоспоживання.

Виходячи з результатів даного тесту та теорії можна зробити висновок про ефективність алгоритму LF в більшості випадків. Однак, існують хмарні системи, які надають приблизно рівні ресурси для всіх заявок на віртуальні машини. У таких системах краще використовувати алгоритм FF, тому що в недовантаженій хмарі цей алгоритм буде рівномірно розподіляти ресурси, і відповідно віртуальні машини будуть отримають більшу продуктивність недовантаженій хмарі. В основному при виборі алгоритму пошуку ресурсів, все ж краще віддати перевагу алгоритму LF.

2.5. Висновки до розділу 2

В цьому розділі розглянуто проблему ресурсного голоду, яка

притаманна багатьом стандартним алгоритмам планування задач. Для її вирішення запропоновано та описано гібридний алгоритм планування, який базується на алгоритмах Round Robin та Simple Job First та дозволяє зменшити час виконання та час очікування класичного витіскального алгоритму Round Robin. Такий алгоритм не страждає від проблеми ресурсного голоду та є більш оптимальним для обробки задач.

Оскільки цей алгоритм відноситься до алгоритмів планування з наперед заданим часом виконання задач, було розглянуто основні алгоритми оцінки часу виконання програмного коду за допомогою функцій працемісткості.

Крім того, було розглянуто основні алгоритми та методи роботи з обчислювальними ресурсам, їх моніторинг та виділення. Було розглянуто переваги та недоліки кожного та виконано порівняльний аналіз. В результаті було обрано алгоритм LF, оскільки він виявився найбільш ефективним.

3. ПРОГРАМНА РЕАЛІЗАЦІЯ СИСТЕМИ

3.1. Загальна структура та архітектура системи

Систему обробки запитів та виконання задач умовно можна поділити на два рівні, це рівень управління та рівень виконання (рис. 14). Рівень управління включає в себе інструменти, що необхідні для керування системою: налаштування прав доступу, ролей, встановлення обмежень та ін., а також прикладний інтерфейс, за допомогою якого сервіси на рівні виконання зможуть мати в наявності усі необхідні параметри для прийняття рішення щодо виконання задачі.

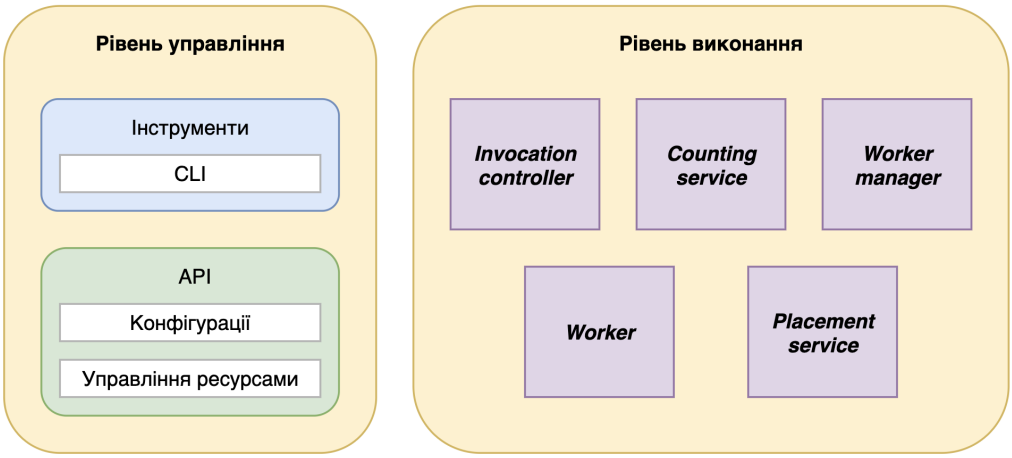


Рис. 14. Загальна структура системи

Рівень виконання складається з таких сервісів:

1. Invocation controller – сервіс контролю за викликами, забезпечує оркестрування синхронних та асинхронних викликів.
2. Counting service – сервіс підрахунку завантаженості, відповідає за моніторинг завантаженості системи, забезпечує дотримання встановлених обмежень при масштабуванні.
3. Worker manager – сервіс керування ресурсами, відповідає за розподілення задач між ресурсами, слідкує за процесом виконання задач, забезпечує масштабування ресурсів.

4. Worker – сервіс керування контейнерами, відповідає за створення та керування контейнерами, забезпечує безпечне середовище для виконання задач.
5. Placement service – сервіс розміщення, даний сервіс відповідає за розміщення контейнерів на ресурсах, для забезпечення їх оптимальної завантаженості.

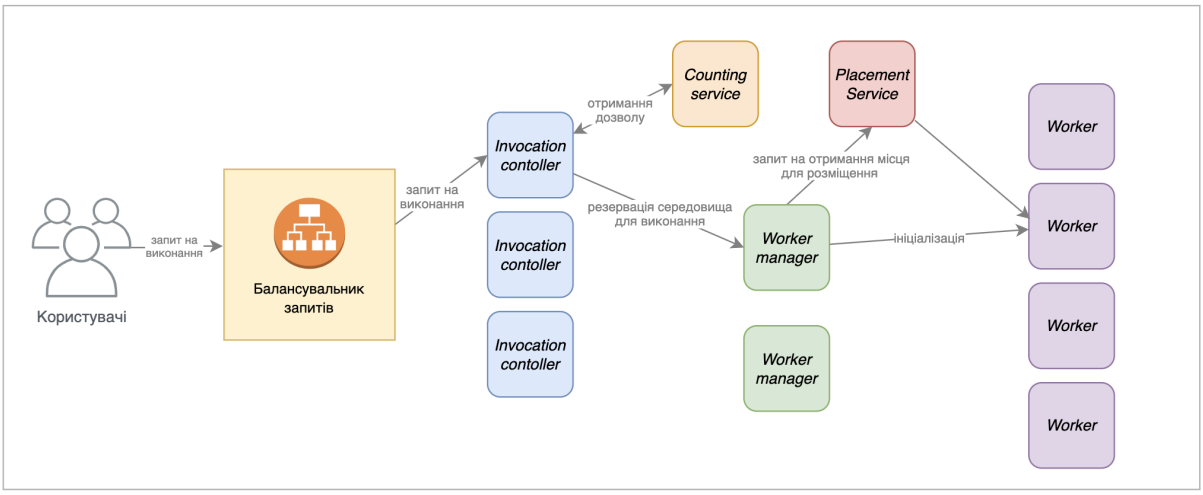


Рис. 15. Взаємодія сервісів системи

На вхід система отримує запит на виконання функції, який може бути створений користувачем чи викликаний у результаті певної події (http запит, виклик за таймером та ін.). У цій роботі ми абстрагуємося від способу виклику, тобто від того, хто, як і за яких умов здійснює запит на виконання задачі. Початковою точкою у системі є запит, який потрапляє до балансувальника . Запит на виконання представляє собою gRPC виклик такому форматі , зображеному на рис. 16.

GRPC – це високопродуктивний фреймворк, розроблений компанією Google для виклику віддалених процедур (RPC), що працює поверх HTTP/2. Він простий у використанні, добре підходить для створення розподілених систем (мікросервісів) та API. Має вбудовану підтримку для балансування навантажень, трасування, аутентифікації та перевірки життєздатності сервісів. Надає можливість створити клієнтські бібліотеки для роботи з

бекендом на 10 мовах програмування. Висока продуктивність досягається за рахунок використання протоколу HTTP / 2 і протокольних буферів (Protocol Buffers) [16].

Protobuf – протокол формату серіалізації, що використовується за замовчуванням для передачі даних між клієнтами та сервером. Використовуючи строгу типізацію полів та бінарний формат для передачі структурованих даних, потребує менше ресурсів. Час виконання процесів серіалізації / десеріалізації значно менший, ніж розмір повідомлень у форматі JSON / XML.

```
message IncomingJobRequest {
  required string token = 1;
  required JobSettings jobSetting = 2;
  required AccountSettings accountSettings = 3;
  required CallerInformation callerInformation = 4;
}
```

Рис. 16. Приклад gRPC повідомлення

Цей виклик проходить через балансувальник запитів, який розподіляє навантаження між сервісами контролю викликів. Для реалізації балансувальнику обрано Envoy Proxy.

Envoy – це високопродуктивний розподілений проксі-сервер (написаний на C ++), призначений для окремих сервісів і додатків, також це комунікаційна шина і «universal data plane», розроблений для великих мікросервісних архітектур «service mesh». При його створенні були враховані рішення проблем, що виникали при розробці таких серверів, як NGINX, HAProxy, апаратних балансувальників навантажень та обласних балансувальників навантажень. Envoy працює разом із кожним додатком та абстрагує мережу, надаючи спільні функції, незалежні від платформ. Коли весь службовий трафік в інфраструктурі проходить через мережа Envoy, це забезпечує легку візуалізацію проблемних областей за допомогою

узгодженої спостережливості, налаштування загальної продуктивності та додавання основних функцій у визначеному місці.

Процес обробки запиту у вигляді блок-схеми зображено на рис. 17.

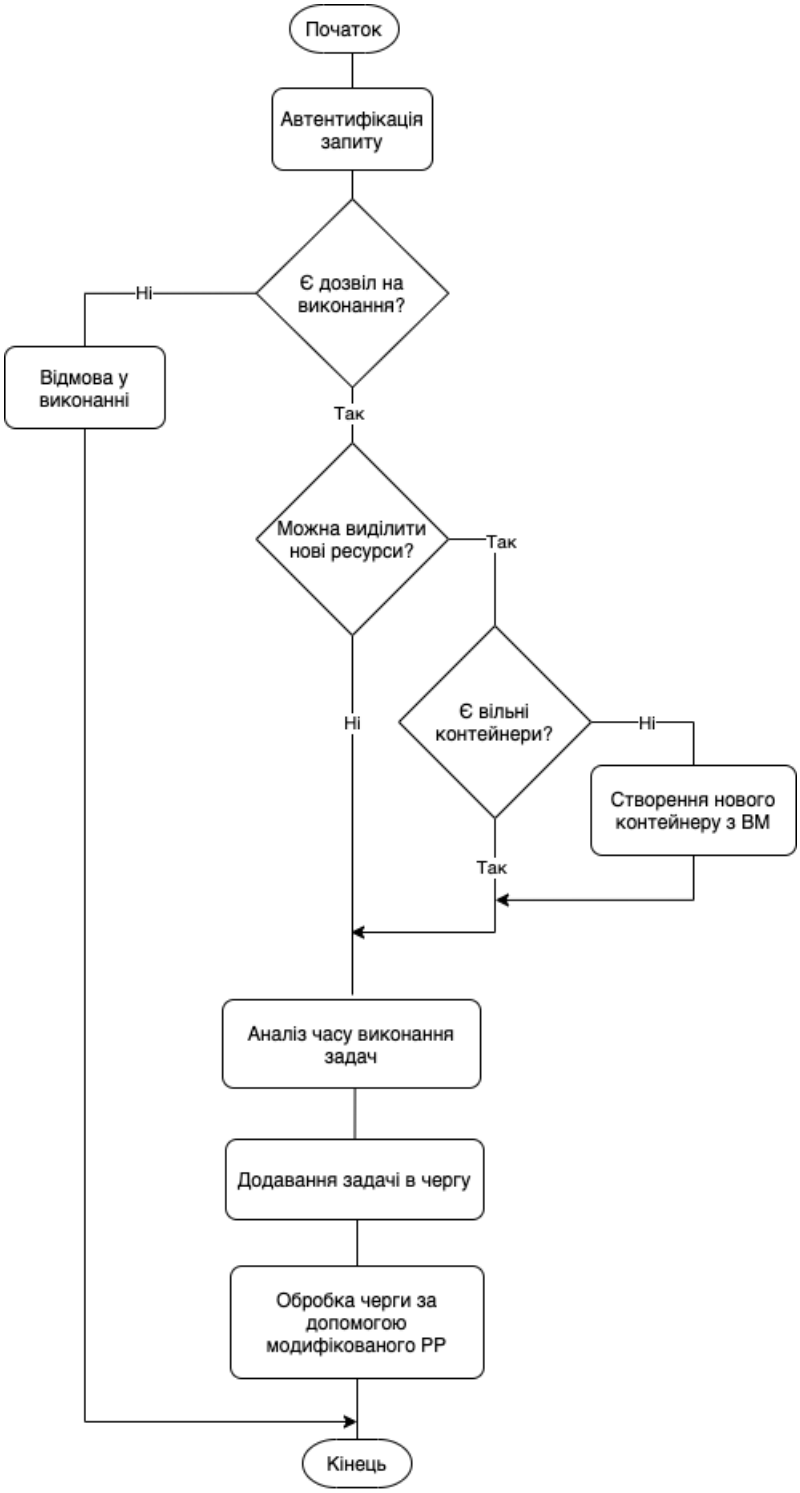


Рис. 17. Блок-схема обробки запиту

Спочатку сервіс контролю викликів (Invocation controller) виконує

автентифікацію запиту, тобто пересвідчується, чи є дозвіл на його виконання. Якщо ні – то буде повернуто відмову у виконанні. Це може статися у таких випадках:

- виклик здійснюється сервісом чи користувачем, який не має відповідних прав, тобто коли їх не дозволено системою створювати дані запити;
- неможливо достовірно встановити, хто здійснює даний запит;
- закінчився час дії токену авторизації і потрібно його оновити.

Якщо обробка запиту можлива, сервіс контролю викликів (Invocation controller) та за допомогою виклику сервісу підрахунку завантаженості перевіряє, чи можливо виділити нові ресурси для обробки запиту. Якщо у поточного користувача ще не досягнуто ліміт за кількістю паралельних ресурсів, йому буде виділено новий ресурс. Для цього сервіс керування ресурсами (Worker manager) спочатку перевіряє, чи є в нього вільні контейнери, щоб обробити цей запит. Якщо є – вони надаються користувачеві, якщо ні – сервіс надсилає запит до Placement service, щоб той виділив нові контейнери.

Після того, як ресурси виділено, Worker manager виконує оцінку часу виконання задачі, після чого розміщує її у черзі та починає оброблювати її таким чином, як описано у п. 3.4.

Для реалізації спілкування між сервісами було використано технологію Linkerd, що є реалізацією service mesh. Service mesh – це виділений рівень інфраструктури, який конфігурується, для забезпечення взаємодії між сервісами системи, що використовується для обробки великої кількості мережевих комунікацій між програмними інтерфейсами застосунків (API) [17].

Він відповідає за надійну доставку запитів через складну топологію сервісів, що створені для роботи у хмарі. Важливою перевагою Linkerd є підтримка HTTP/2, що в свою чергу дозволить використання протоколу gRPC.

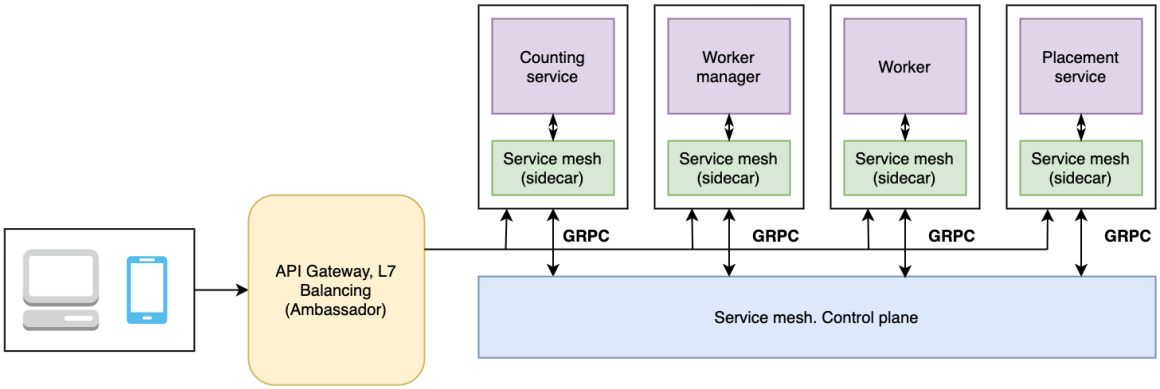


Рис. 18. Схема комунікації сервісів

3.2. Опис та реалізація сервісу контролю за викликами

Зоною відповідальності сервісу Invocation controller є автентифікація запитів та перевірка дозволів їх виконання. Він пересвідчується у тому, що тільки авторизовані запити проходять далі до системи та мають можливість викликати виконання задачі.

Порядок обробки запиту:

1. Перевірка прав на виконання запиту.
2. Завантаження метаданих задачі, що створюється.
3. Завантаження змінних середовища та обмежень на виконання.
4. Перевірка кількості одночасно запущених задач та співставлення з обмеженнями за допомогою сервісу підрахунку.

Для перевірки прав на виконання запити повинні містити заголовок з токеном, у якому закодовано інформацію про сутність, що здійснює запит. Для авторизації використовується JWT – JSN Web token. Токен JWT (рис. 19-20) складається трьох частин: заголовка (header), корисного навантаження (payload) та підписи або даних шифрування (signature). Перші два елементи – це об’єкти JSON визначеної структури. Третій елемент обчислюється на основі перших і залежить від обраного алгоритму. Токени можуть бути перекодовані в компактному представленні (JWS / JWE Compact Serialization): до заголовка та корисного навантаження

застосовується алгоритм кодування Base64-URL, після чого додається підпис та всі три елементи розділяються крапками.

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.ey
JhY2NvdW50SWQiOiJhYzQyYTUzYi0yNmYwLTQyZ
DAtOGRiZi1iZjViMDNjNDUzZjciLCJjYWxsZXJJ
ZCI6IjhkODcxOTQ5LWE5YzQtNGM5Zi1iZmUxLTR
mYzE3MjBjZTlmOCIsImlhdCI6MTU5NTIzOTAyMn
0.dhRPKw6QnYTkFYvqf3zKx-
aLuYi3rUwL0b1yTRkWFmw
```

Рис. 19. Приклад JWT токену

```
{
    "accountId": "ac42a53b-26f0-42d0-8dbf-bf5b03c453f7",
    "callerId": "8d871949-a9c4-4c9f-bfe1-4fc1720ce9f8",
    "iat": 1595239022
}
```

Рис. 20. Приклад закодованих даних у другій частині JWT токену

3.3. Опис та реалізація сервісу підрахунку завантаженості

Сервіс підрахунку завантаженості (Counting service) займається моніторингом рівня завантаженості системи, пересвідчується у дотриманні усіх обмежень, що були встановлені користувачем на кількість одночасно запущених задач та здійснює контроль за масштабуванням. Через нього проходять усі запити на створення ресурсів для перевірки можливості масштабування. На основі інформації, переданої у тілі запиту, сервіс підрахунку завантаженості перевірить поточний стан системи, кількість вже запущених задач та порівняє його з обмеженнями встановленими користувачем. У випадку, якщо кількість функцій користувача не досягла зазначеного обмеження, то сервіс дозволить виділення нового ресурсу,

інакше сервіс поверне інформацію, що максимальне значення паралельних ресурсів вже використовується. Цей сервіс має бути швидким, еластичним та не повинен створювати додаткових затримок, щоб не зменшити продуктивність системи, адже він викликається з кожним запитом.

Для збереження інформації щодо поточних запущених задач було обрано СКБД Aerospike, що має тип сховища «ключ-значення», а також реалізує без схемну модель даних. Важливою особливістю Aerospike є підтримка розподілу розділів, що дозволить масштабування, оскільки даний сервіс буде високонавантажений. Ще одним важливим фактором вибору цієї СКБД є підтримка реплікації розділів даних та автоматичне ребалансування, що дозволить сервісу бути стійким до відмови.

Для реалізації обробників запитів до сервісу було обрано програмну платформу Node.js. Її перевагою є модель вводу-виводу, що керується подіями та є неблокувальною , тому вона добре підходить для обробки великої кількості запитів та зможе забезпечити мінімальний час відповіді. Обробники отримують запити із повідомленнями у protobuf форматі, десеріалізують ці повідомлення на основі proto моделі та отримують повідомлення у форматі JSON.

```
service CountingSevice {
  rpc checkConcurrency(CheckConcurrencyRequest) returns (CheckConcurrencyResponse) {}
  rpc notify(ChangeConcurrencyRequest) returns (ChangeConcurrencyResponse) {}
}
```

Рис. 21. Інтерфейс Counting service

Сервіс надає дві процедури для віддаленого виклику (RPC) для інших сервісів:

- checkConcurrency – метод для отримання дозволу на виділення нових ресурсів;
- notify – метод для сповіщення сервісу підрахунку завантаженості, щодо змін статусу ресурсів.

3.4. Опис та реалізація сервісу керування ресурсами

Зоною відповідальності цього сервісу є виділення нових ресурсів, тобто створення нових віртуальних машин, а також планування та розподілення задач між існуючими ресурсами у системі. Він викликається сервісом контролю викликів та передає інформацію, щодо задачі, яку потрібно виконати, а також інформацію про дозвіл на створення нових ресурсів. Якщо був отриманий запит, якому було попередньо надано дозвіл на виділення нової віртуальної машини для виконання, то сервіс керування ресурсами перевірить наявність вільних віртуальних машин серед ресурсів, якими він керує. Якщо сервіс не матиме вільних віртуальних машин для виконання, то він виконає запит до сервісу розміщення ресурсів для отримання віртуальної машини у своє розпорядження. Після чого він помістить задачу до черги для її подальшої обробки гібридним алгоритмом, що був запропонований у підрозділі 2.3.2.

Обробка черги задач гібридним алгоритмом

Перед початком обробки поточну чергу буде зафіксовано та створено нову чергу для прийому задач, тобто усі нові задачі, що будуть надходити будуть потрапляти до цієї нової черги. Далі алгоритм буде працювати з чергою, що була зафіксована на попередньому етапі. Приклад початкової черги:

Рис. 22. Початкова черга задач

Першим кроком обробки є сортування черги задач за зростанням часу, необхідного для їх обробки (рис. 23).

Рис. 23. Відсортована черга задач

Наступним кроком алгоритм виділить більш пріоритетну чергу для обробки, для цього відсортована черга поділяється на дві черги за формулою (8-9). На рис. 24-25 зображено дві новоутворені черги.

Рис. 24. Пріоритетна черга задач

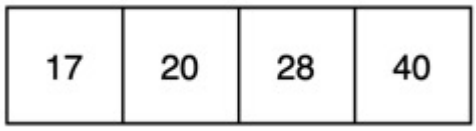


Рис. 25. Залишкова черга задач

Після чого алгоритм опрацьовує ці черги за допомогою модифікованого алгоритму RR з динамічним квантом часу. Спочатку він обробить першу, більш приорітетну чергу за два проходи. Перед обробкою конкретної задачі сервіс спочатку пересвідчиться у наявності збереженого стану виконання і якщо для задачі є збережений стан виконання, то його буде відновлено зі збереженого знімку, таке можливо якщо задача вже виконувалася, проте була перервана. На рис. 24 зображена пріоритетна черга до початку обробки, а на рис. 26 та 27 зображена пріоритетна черга після першої та другої ітерації відповідно.

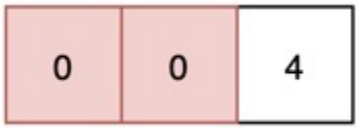


Рис. 26. Пріоритетна черга після 1-ї ітерації

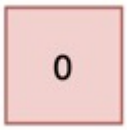


Рис. 27. Пріоритетна черга після 2-ї ітерації

Далі алгоритм робить один прохід другою чергою. На рис. 25 зображена друга черга до початку обробки, а на рис. 28 – після.

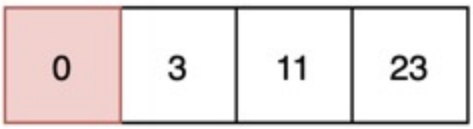


Рис. 28. Залишкова черга після 1-ї ітерації

Після цього залишкові задачі з цієї черги будуть додані до нової черги, у яку додавались усі нові задачі протягом ітерації і алгоритм почне обробку цієї нової.

Оскільки алгоритм виділяє фіксований час на виконання кожній задачі протягом однієї ітерації, то у випадку, коли задачі необхідно більше часу на виконання, ніж було виділено, вона буде перервана по завершенню виділеного часу. У такому випадку сервіс робить знімок стану виконання віртуальної машини, на якій виконується задача, та зберігає його, за допомогою методів віддалених викликів, що надає сервіс керування контейнерами.

3.5. Опис та реалізація сервісу керування контейнерами

Worker – важливий компонент архітектури системи, що забезпечує безпечне середовище для виконання задач. Сервіс відповідає за створення та керування колекцією контейнерів з середовищами для виконання задач, встановлює обмеження для виконання задачі, такі як кількість оперативної пам'яті, кількість ядер процесору та ін., завантажує код задачі та встановлює його. Також даний сервіс надає інтерфейс для створення нової віртуальної машини, створення знімку стану віртуальної машини, а також відновлення стану віртуальної машини зі зліпку. На рис. 29 зображено інтерфейс сервісу описаний у protobuf форматі.

```
service WorkerService {
  rpc createVM(CreateVMRequest) returns (CreateVMResponse) {}
  rpc makeSnapshot(MakeSnapshotRequest) returns (MakeSnapshotResponse) {}
  rpc restoreFromSnapshot(RestoreSnapshotRequest) returns (RestoreSnapshotResponse) {}
}
```

Рис. 29. Інтерфейс Worker service

Технології реалізації ізоляції контейнерів

Контейнеризація у Linux забезпечується за допомогою групування функцій ядра у окремі групи, що дозволяє створити додатковий рівень ізоляції. Основні механізми для реалізації ізоляції:

Cgroups – механізм, що забезпечує контроль над тим, скільки ресурсів таких як оперативна пам'ять, кількість ядер процесора ті ін. буде надано процесу, а також усім процесам та потокам, що він породжує. За допомогою цього механізму ми можемо забезпечити операційну ізоляцію для певної задачі, оскільки процеси, що виконуються всередині певної групи не можуть вибратися за її межі.

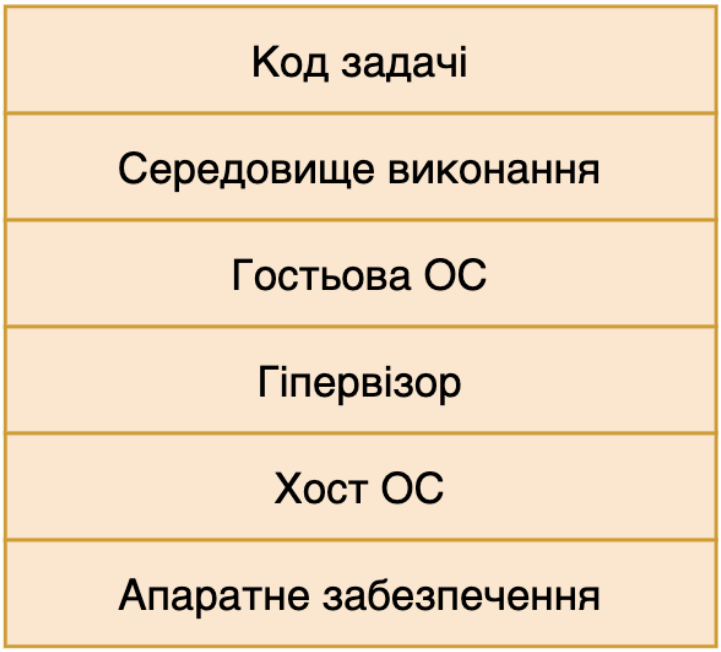


Рис. 30. Структура рівнів архітектури сервісу

Namespaces – для кожної задачі створюється простір імен, у рамках якого будуть створені усі ідентифікатори (process id, group id, user id).

Seccomp – файервол для ядра Linux. Ядро має велику кількість системних викликів, що надають інтерфейс для можливих функцій ядра, таких як відкриття сокетів, відкриття файлу, зчитування та запис до файлу та ін. Seccomp дозволяє тільки ті системні виклики з тими параметрами, які необхідні задачі і забороняє виконання непередбачуваних системних викликів.

Технології віртуалізації та емуляції пристроїв

Наступний рівень ізоляції – віртуалізація та емуляція пристроїв [18]. Це використання технологій віртуалізації, що закладені у апаратне забезпечення, наприклад такі як VTX на процесорах Intel, щоб працювати з апаратним забезпеченням так, як ніби існує декілька однакових елементів такого апаратного забезпечення замість одного. Це все регулюється гіпервізором або монітором віртуальної машини (VMM). Гіпервізор представляє собою рівень програмного забезпечення, що знаходиться під рівнем гостьових операційних систем та створює для них віртуальне

апаратне забезпечення, а також забезпечує синхронізацію для оптимального використання справжнього апаратного забезпечення.

Firecracker – це технологія віртуалізації з відкритим кодом, яка була розроблена спеціально для створення та керування багатокористувацькими контейнерами. Firecracker є монітором віртуальних машин, що використовує віртуальну машину на базі ядра Linux (KVM) для створення та управління віртуальними мікро-машинами, що в свою чергу забезпечує посилений захист у порівнянні з традиційними віртуальними машинами. Це гарантує, що навантаження від різних кінцевих користувачів можуть безпечно працювати на одній машині. Система має мінімалістичний дизайн, що запобігає використанню зайвих пристроїв, а також виключає усі несуттєві функціональні можливості , що дозволяє зменшити використання пам’яті, а це є дуже важливим фактором, адже кількість таких віртуальних мікро-машин зазвичай є великою. Такий підхід покращує безпеку, зменшує час запуску та збільшує перевикористання апаратного забезпечення. На додаток до мінімалістичної моделі системи, Firecracker також прискорює завантаження ядра та забезпечує мінімалістичну конфігурацію гостьового ядра. Має можливість створювати до 150 віртуальних мікро-машин за секунду на хост системі.

Основні особливості:

- проста гостьова модель, для усіх гостей забезпечено лише мінімум пристроїв;
- ізолювання процесу Firecracker за допомогою cgroups та seccomp BPF, а також обмеженого набору системних викликів;
- статична прив’язка процесу для його запуску в умовах ізоляції від оточення хост-системи.

У разі повної віртуалізації, гостьова операційна система працює поверх гіпервізора, який встановлений на голому апаратному забезпеченні. Гостьова система не знає, що вона в даний момент є віртуалізованою і тому в цій конфігурації не потрібно ніяких змін в її роботі. Навпаки, при

паравіртуалізації, гостьова операційна система не тільки знає, що вона запущена на гіпервізорі, а й містить код, який підвищує ефективність взаємодії гостьової системи з гіпервизором. При повній віртуалізації гіпервізор повинен емулювати апаратне забезпечення пристрою, що відбувається на найнижчому рівні обміну даними (наприклад, для мережевого драйвера). Хоча завдяки абстракції зрозуміло, як виконувати емуляцію, вона в більшості випадків неефективна і дуже складна. У разі паравіртуалізації гостьова система і гіпервізор можуть працювати спільно з метою підвищення ефективності цієї емуляції. Недоліком підходу з використанням паравіртуалізації є те, що операційна система знає, що вона є віртуалізуваною і в її роботу потрібно вносити зміни. Для нашої системи паравіртуалізація буде більш переважним варіантом та дозволить отримати виграш у продуктивності.

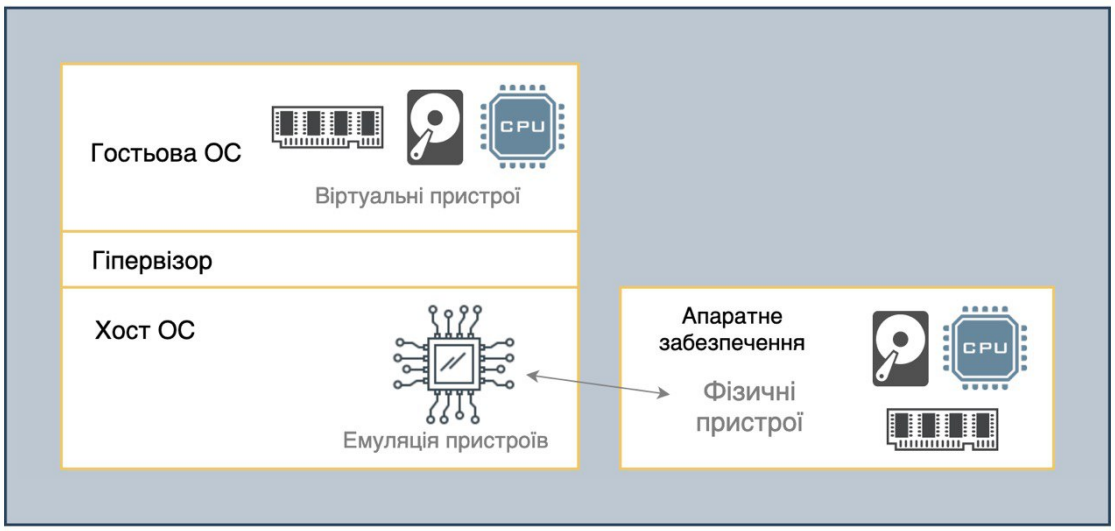


Рис. 31. Схема емуляції апаратного забезпечення

Virtio – фреймворк, який забезпечує абстрактний рівень над апаратним забезпеченням у гіпервізорі, що реалізовано на принципах паравіртуалізації. Він визначає спосіб взаємодії для гостьової системи та хост системи та надає абстрактний інтерфейс для роботи з пристроями. Даний фреймворк дозволяє оптимізувати віртуалізацію, за допомогою

зменшення кількості перемикань між гостьовою та хост системами. Він збирає необхідну інформацію у пам’яті, а потім повідомляє гіпервізор про необхідність обробки цих даних після чого ці дані надходять до емульованого пристрою, який далі у свою чергу відправить це до справжнього апаратного забезпечення.

Даний сервіс надає інтерфейс, за допомогою якого сервіс виділення ресурсів зможе створювати та конфігурувати віртуальні машини, а також інтерфейс, за допомогою якого можна спостерігати за станом цього сервісу. Взаємодія з самим Firecracker відбувається за допомогою API через файл сокет. Між інтерфейсом який надає даний сервіс та рівнем роботи з гіпервізором знаходиться рівень трансформації команд, що викликає необхідні команди гіпервізора на основі отриманих вхідних даних. Крім того, цей рівень дозволяє викликати тільки певний набір команд з системного інтерфейсу гіпервізора, що забезпечує безпеку системи. Коли сервіс отримує команду createVM через протокол gRPC, він перетворює параметри, що були отримані у повідомлені на параметри, що приймає Firecracker, після чого робить запит через сокет у форматі, який підтримує Firecracker. Завдяки цьому забезпечується рівень абстракції над інтерфейсом Firecracker та зменшується зв’язаність систем, що дозволить за необхідності змінити поточну реалізацію гіпервізора (Firecracker) на будь-яку іншу. На рис. 32 зображено приклад команди для конфігурації віртуальної машини Firecracker.

```
$ curl --unix-socket /tmp/firecracker.sock -i \
    -X PUT "http://localhost/machine-config" \
    -H "accept: application/json" \
    -H "Content-Type: application/json" \
    -d "{
        \"vcpu_count\": 1,
        \"mem_size_mib\": 512
    }"
```

Рис. 32. Приклад команди для конфігурації віртуальної машини Firecracker

Після створення машини з необхідними параметрами, її буде запущено за допомогою команди зображеної на рис. 33.

```
curl --unix-socket /tmp/firecracker.sock -i \
  -X PUT "http://localhost/actions" \
  -H  "accept: application/json" \
  -H  "Content-Type: application/json" \
  -d "{
      \"action_type\": \"InstanceStart\"
   }"
```

Рис. 33. Приклад команди для запуску віртуальної машини Firecracker

3.6. Опис та реалізація сервісу розміщення ресурсів

Placement service відповідає за створення та виділення віртуальних машин, використовуючи для цього інтерфейс, який надає Worker service та передає ці віртуальні машини у керування сервісу Worker manager. Він, як і інші сервіси, надає інтерфейс для викликів за допомогою gRPC протоколу. На рис. 34 зображено інтерфейс сервісу розміщення ресурсів.

```
service PlacementService {
  rpc getVM(GetVMRequest) returns (GetVMResponse) {}
}
```

Рис. 34. Інтерфейс Placement service

Даний сервіс при виділені існуючого вільного ресурсу враховує такі фактори як: кількість ядер, обсяг оперативної пам’яті, обсяг сховища даних та ін. У випадку відсутності готових віртуальних машин, що підходять за параметрами, він створює нову віртуальну машину з необхідними параметрами за допомогою інтерфейсу, що надає Worker Service. У своїй реалізації використовує алгоритм Best fit з бібліотеки, який на основі отриманих параметрів підбирає найбільш відповідну віртуальну машину.

3.7. Висновки до розділу 3

Для реалізації системи було розроблено архітектуру, що складається з 5 основних сервісів: сервіс підрахунку завантаженості (Counting service), сервіс контролю за викликами (Invocation controller service), сервіс керування ресурсами (Worker manager), сервіс керування віртуальними машинами (Worker) та сервіс розміщення ресурсів (Placement service). Кожен із сервісів інкапсулює частину функціональності системи та надає інтерфейс для її використання:

1. Invocation controller – перевірка прав на виконання запитів.
2. Counting service – моніторинг рівня завантаженості системи.
3. Worker – створення віртуальних машин.
4. Worker manager – керування процесом виконання задач.
5. Placement service – виділення ресурсів під виконання.

Для спілкування сервісів між собою було виділено окремний транспортний рівень (Service mesh), який відповідає за маршрутизацію повідомлень між сервісами та обробку помилок транспортного рівня. При цьому для спілкування між сервісами було обрано протокол gRPC, завдяки таким факторам:

1. Підтримка HTTP2, що покращує швидкість обміну повідомленнями.
2. Використання proto моделей, що надає переваги статичної типізації.
3. Вбудована серіалізація та десеріалізація повідомлень.

4. ТЕСТУВАННЯ РОЗРОБЛЕНИХ ЗАСОБІВ

4.1. Тестування алгоритму планування задач

Для перевірки практичної цінності описаного гібридного алгоритму для задачі балансування навантаження між ресурсами, було проведено симуляцію інфраструктури хмарної системи за допомогою інструменту CloudSim.

Платформа CloudSim – це абстрактний та масштабований засіб симуляції, що надає можливість проводити повноцінне моделювання і симуляцію хмарних обчислювальних систем і інфраструктур [10, 11]. Він є розширенням базової функціональності платформи GridSim та забезпечує моделювання сховищ даних, використання веб-сервісів, розподіл ресурсів між віртуальними машинами та інше [19].

Для проведення симуляції було реалізовано запропонований алгоритм за допомогою CloudSim версії 3.0. Для реалізації інфраструктури за основу було обрано сервера HP ProLiant G4. Ці сервери мають здатність виконувати 1860 мільйонів операцій на секунду на кожне ядро процесору. Усі створені віртуальні машини мають одне ядро, 128 МБ оперативної пам’яті.

Для початку роботи з CloudSim спочатку потрібно виконати його ініціалізацію за допомогою команди *CloudSim.init()*. Після цього необхідно створити сутність провайдера ресурсів, який може бути створено за допомогою відповідної команди *createDatacenter*(). Після створення провайдера ресурсів необхідно створити брокера, який буде надсилати задачі для виконання до провайдеру ресурсів, а також створити віртуальні машини. У випадку створення декількох віртуальних машин необхідно надати список цих машин брокеру. Симуляція починається за допомогою команди *CloudSim.startSimulation*(). Основні класи якими оперує CloudSim для створення інфраструктури: Cloudlet, Cloudlet Scheduler, Datacenterer, DatacenterBroker, Host, Vm, VmScheduler.

Для порівняння існуючих алгоритмів та запропонованого гібридного

алгоритму їх було перевірено на наборі із 1000 випадково згенерованих задач, кожна з яких характеризується часом сплеску (burst time) у мілісекундах. Для проведення порівняння на цих даних у системі CloudSim було також реалізовано алгоритму FCFS, Simple Job First та Round Robin з константним значенням кванту часу [20].

Порівняння алгоритмів проводиться за наступними критеріями [21, 22]:

1. Ефективність використання CPU. Він має бути постійно завантажений, з корисною роботою 100%.
2. Час оберту – час від надсилання запиту до повного його виконання.

На рис. 35 зображено середній відсоток завантаження кожного CPU в залежності від кількості віртуальних машин. Зі збільшенням кількості віртуальних машин, середнє використання кожної з них зменшилось. Однак запропонований алгоритм мав найвищий середній коефіцієнт використання ядра для всіх віртуальних машин, що є бажаною поведінкою. Можна побачити, що найгірше з оптимальним використанням ресурсів справляється SJF, причини чого були описані у розділі 2.

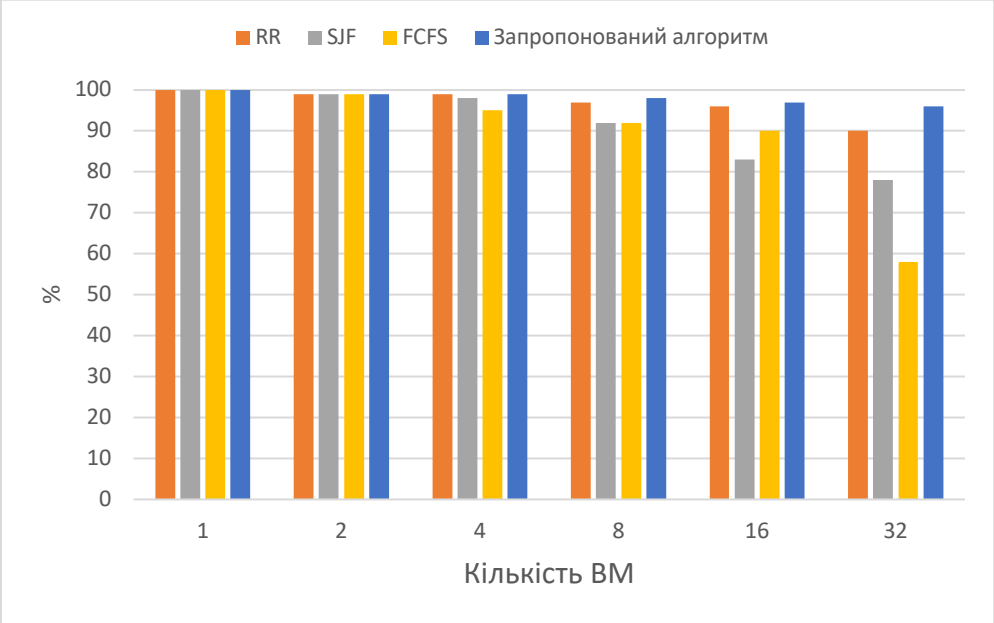


Рис. 35. Ефективність використання CPU для різної кількості ВМ

Порівняємо час виконання 1000 задач для різної кількості віртуальних машин (рис. 36). Як видно з графіку, алгоритм Round Robin з фіксованим квантом часу завжди займає більше часу для обробки черги, а алгоритм SJF навпаки виконує усі задачі швидше за інші. Середній час виконання задач зменшується зі збільшенням кількості віртуальних машин.

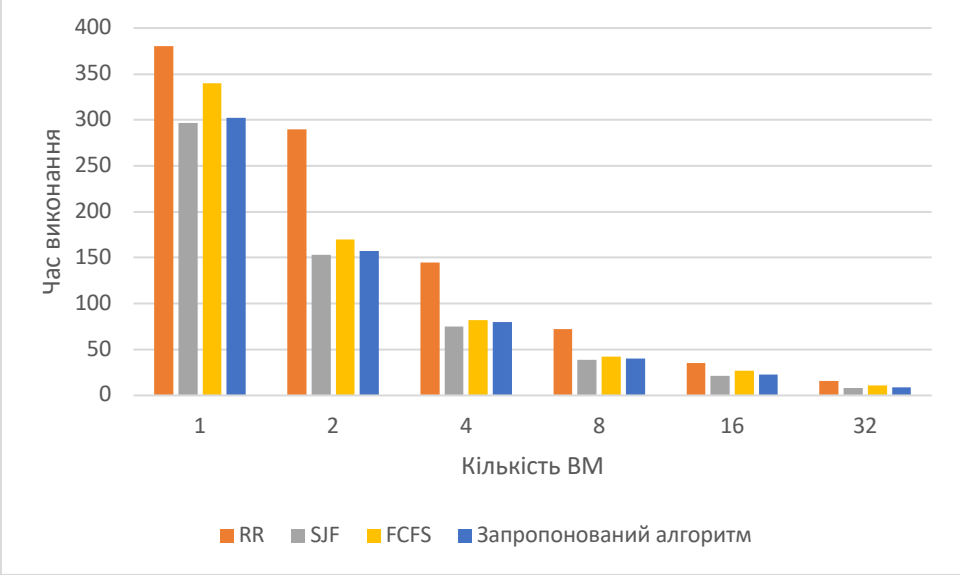


Рис. 36. Середній час виконання 1000 запитів для різної кількості віртуальних машин

Розглянемо ще середній час виконання іншої кількості запитів для восьми віртуальних машин (рис. 37).

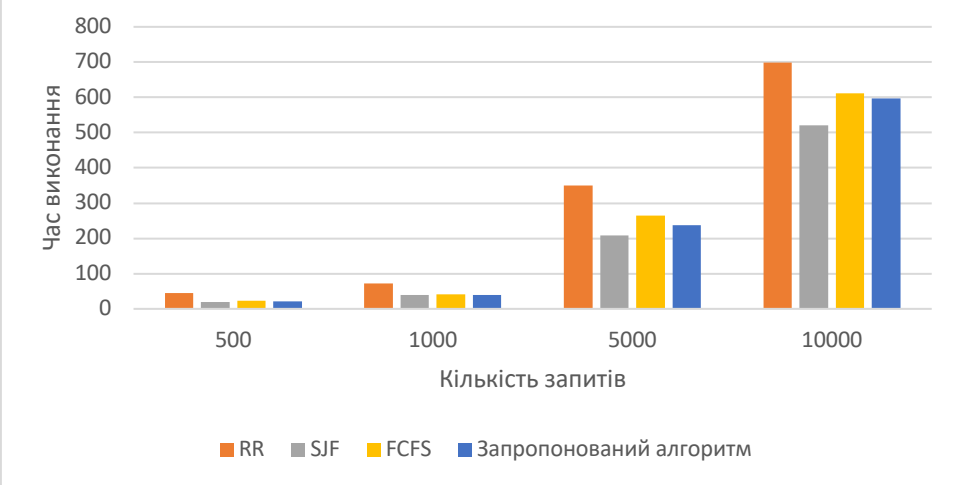


Рис. 37. Середній час виконання запитів для восьми віртуальних машин

Отже, запропонований алгоритм суттєво зменшує час обробки запитів у порівнянні з алгоритмом Round Robin з фіксованим квантом часу. При цьому даний алгоритм не суттєво програє у часі виконання алгоритму SJF, але суттєво виграє у нього в ефективності використання ресурсів.

4.2. Тестування системи

Для цілісного тестування системи було використано утиліту hey [23]. Hey – це сучасна система для тестування з навантаженням, яка надає можливість надсилати запити до розробленої системи. Утиліта має консольний інтерфейс та дозволяє конфігурувати параметри для тестування такі як: кількість одночасних запитів, загальну кількість запитів, максимальний час очікування на відповідь та ін. В утиліті задається посилання, у результаті запитів до якого будуть створені функції, що оброблять дану подію.

Для тестування було обрано функцію сортування масиву вставками, де довжина масиву випадковим чином генерувалась в діапазоні від 1000000 до 5000000 елементів, емулюючи таким чином різні функції. Для тестування було заплановано всього 1000 запитів до системи, при цьому одночасно надсилалося 5 запитів для перевірки паралельної обробки.

На рис. 38 зображено звіт утиліти про проведене тестування для 10000 запитів (функцій).

Також варто зазначити, що можливо отримати звіт у csv форматі, де буде надано інформацію про кожен запит окремо. Такий формат відповіді зручний тим, що його можна використовувати для подальшого детального аналізу. На рис. 38 зображено підсумок виконання тестування, де заначено: загальний час, який витрачено на виконання усіх 1000 запитів, час найшвидшої відповіді, найповільніший час відповіді, середній час відповіді, а також середнє значення виконаних запитів/сек.

```
→  diploma hey -t 50  -c 5 -n 1000 http://localhost/hello

Summary:
  Total:        1125.8082 secs
  Slowest:      13.7845 secs
  Fastest:      0.0315 secs
  Average:      5.3493 secs
  Requests/sec: 0.8883

  Total data:   18150 bytes
  Size/request: 18 bytes
```

Рис. 38. Звіт про виконання тестування

На рис. 39 зображено гістограму часу відповідей для задач, на якій зображено нижні границі проміжку часу відповідей, з якої видно, що для більшої частини задач час відповіді знаходився у діапазоні від 2.7 секунд до 9.65 секунд.

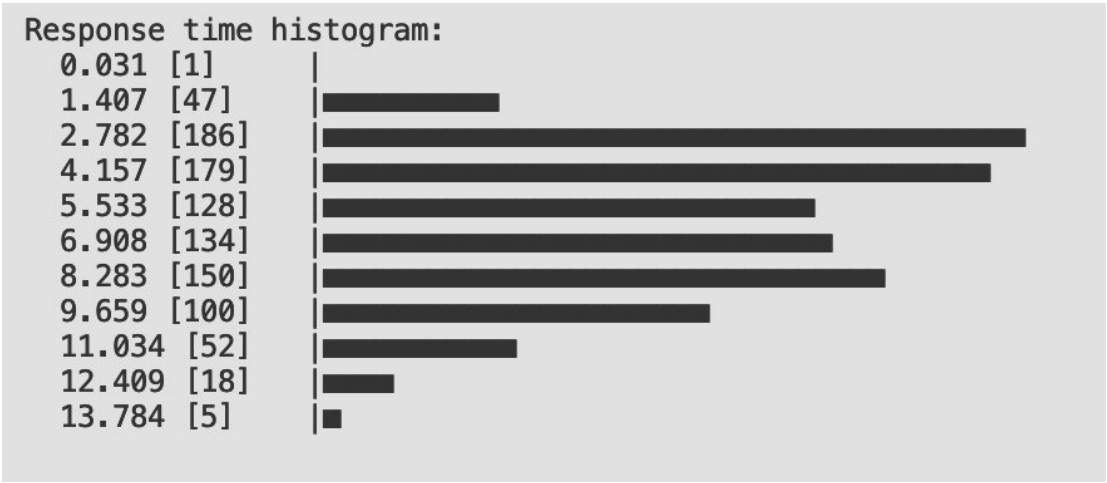


Рис. 39. Гістограма часу відповідей

4.3. Висновки до розділу 4

Для тестування гібридного алгоритму використано платформу CloudSim 3.0 за допомогою якої було згенеровано задачі для тестування, а також побудовано інфраструктуру для порівняння існуючих алгоритмів з запропонованим. Для тестування усі алгоритми були реалізовані на мові

Java з використанням класів, що надає бібліотека CloudSim, що дозволило провести симуляцію для усіх вищезазначених алгоритмів. Оскільки усі алгоритми було протестовано на однакових даних та за однакової інфраструктури, вдалося порівняти отримані результати. Запропонований алгоритм зміг суттєво зменшити час обробки задач у порівнянні з алгоритмом Round Robin, який має фіксований квант часу. Також даний алгоритм дозволяє зменшити явище ресурсного голоду у порівняні з алгоритмом Simple Job First, оскільки є більш справедливим до ресурсоємних задач.

Для перевірки всієї системи було проведене тестування з навантаженням за допомогою утиліти hey на різному наборі задач. Завдяки цьому вдалося пересвідчитися в працездатності системи під навантаженням та оцінити її поведінку.

5. РОЗРОБЛЕННЯ СТАРТАП-ПРОЄКТУ

5.1. Опис проблеми

Розглянемо можливість створення стартап-проєкту хмарного сервісу.

Національний інститут стандартів і технологій США описує хмарні обчислення [10] як модель мережевого доступу до загального набору параметрів та обчислювальних ресурсів (наприклад, мережевих каналів, процесорів, пам’яті, пристроїв зберігання даних, додатків і сервісів), які можуть швидко виділятися за запитом користувача при мінімальних зусиллях з боку хмарного провайдера.

При цьому головна особливість хмарних обчислень і їх відмінність від корпоративних IT-систем полягає в тому, що користувач, запитуючи і отримуючи інформацію або інші хмарні сервіси зі свого ПК, абсолютно не уявляє, де вони фізично розташовані і яким способом виконуються.

Таким чином, «хмара» – це певна модель збереження та використання сервісів та ресурсів, а також їх адміністрація.

При цьому отримуємо ряд переваг над класичним використанням офісних хостів:

1. Ефективність. Ресурси не залежать від ОС і їх географічного розташування. Це забезпечує реальну економію коштів завдяки легкому масштабуванню ресурсів за потребами (балансуванню навантаження).
2. Безпека. Хоч хмарні системи славляться низьким рівнем безпеки даних, це досить легко спростувати. Захист даних у хмарі забезпечується не лише за допомогою звичаного шифрування, а й на локальному рівні шляхом впровадження в віртуальні машини ряду правил, важливих для захисту інформації з високим пріоритетом.
3. Гнучкість. По-перше, всі ресурси, ПЗ і технічне забезпечення можуть бути перебудовані в нові сервіси та послуги практично

миттєво. По-друге, ресурси можна легко масштабувати шляхом планування завдань та методом балансування навантаження.

4. Надійність. У хмарі реалізований високий рівень резервування даних, при цьому для збереження копії і поновлення системи виділяються необхідні ресурси. Цей процес повністю автоматичний, що також істотно виділяє перевагу над офісними серверами.
5. Автоматизація. Програмне забезпечення для управління ресурсами хмарної системи налаштоване на автоматичне виконання своїх функцій, динамічно виділяючи необхідні ресурси користувачеві для їх використання. Це збільшує ефективність виділення ресурсів та сприяє зменшенню кількості обслуговуючого сервери IT-персоналу.
6. Легкість доступу. Користувачам надається значно більше додатків, інформації, ресурсів і послуг, ніж це може забезпечити звичайний сервер. Зазвичай, доступ надається через будь-який браузер.
7. Оптимізація. Хмарна система контролюється як цільна система, через це кожен користувач отримує можливість власної оптимізації потрібних ресурсів за рахунок оптимального поєднання можливостей сервісів, продуктивності і вартості.

Отже, хмарні обчислення призводять до кращого способу розподілення ресурсів. При цьому спожитими ресурсами і сервісами хмарної системи легко можна керувати та встановлювати ліміти таким чином, щоб користувачі сплачували лише за фактично використані потужності.

У порівнянні з корпоративним сервером, перевага «хмари» ще полягає в тому, що будь-який користувач отримує «хмарні» послуги за запитом, при цьому регулювати їх обсяги може самостійно. Всі ресурси «хмари» при наявності інтернету, доступні з будь-якого місця планети в будь-який час доби.

Запропонована система буде використовувати модель SaaS, яка вважається найпоширенішою в світі, так як її використовують більшість користувачів інтернету. Кожного дня ми користуємося продуктами цієї моделі, навіть не усвідомлюючи, що вони розміщені в хмарах. Модель SaaS працює за досить простим принципом: програми і сервіси розробляє і адмініструє провайдер, розміщує їх в хмарі та надає доступ користувачу через браузер або додаток на його комп'ютері. Клієнт обирає послугу, сплачує її (необов'язково), а технічною підтримкою програм займається провайдер.

Будь-які користувачі, які мають потребу у виконанні програмного коду, стикаються з потребами у сховищі даних та обчислювальних потужностях. Деякі користувачі потребують просто місце для збереження їх даних, з можливостю легко розширювати своє сховище. Вони зацікавлені в тому, щоб сплачувати лише за фактично зайняте місце, при цьому бути впевненими у надійності сховища та мати можливість доступатись до своїх даних з будь-якого місці та пристрою.

Деякі користувачі, наприклад, програмісти, дослідники та компанії зацікавлені в хостингу для свого програмного коду. Їм потрібні обчислювальні потужності, але купувати власні сервери може бути незручним через декілька причин. По-перше, їх важко розгортати. Це вимагає певних коштів для придбання якісної апаратури, найму спеціалістів для налаштування, підтримання стабільного електропостачання. При цьому, якщо придбаних серверів в певний момент часу стане вже замало, необхідно буде купувати ще більше серверів, що робить масштабування досить складним.

По-друге, деяким користувачам недоцільно створювати фізичні сервери, оскільки вони мають потребу виконувати ресурсоємні операції досить рідко. Таким чином, якщо вони придбають необхідні апаратні ресурси, більшу частину часу вони будуть простоювати, роблячи це вкладання коштів марним.

Методи планування задач та виділення ресурсів в хмарних системах є ключовою особливістю, яка відрізняє провайдерів один від одного. Існуючі алгоритми обробки черги задач мають певні недоліки, які погіршують роботу системи. В деяких алгоритмах задачі очікують в черзі надто довго на початок обробки. Інші алгоритми страждають від того, що ціла черга оброблюється надто довго, хоча кожна задача починає оброблюватись швидко. В таких алгоритмах також присутня дуже велика кількість перемикань контексту, що призводить до неоптимального використання ресурсів. Якщо запити оброблюються послідовно, може виникати також проблема ресурсного голоду, у випадку коли певна задача заблокувала потік виконання. Крім того, часто можуть виникати ситуації, коли на обробку запитів виділено декілька віртуальних машин, проте CPU кожної з них не завантажений повністю. В цьому випадку більш оптимальним було б використовувати меншу кількість віртуальних машин, але щоб кожна з них була завантажена повністю. Ці проблеми схематично подано у вигляді дерева на рис. 40.

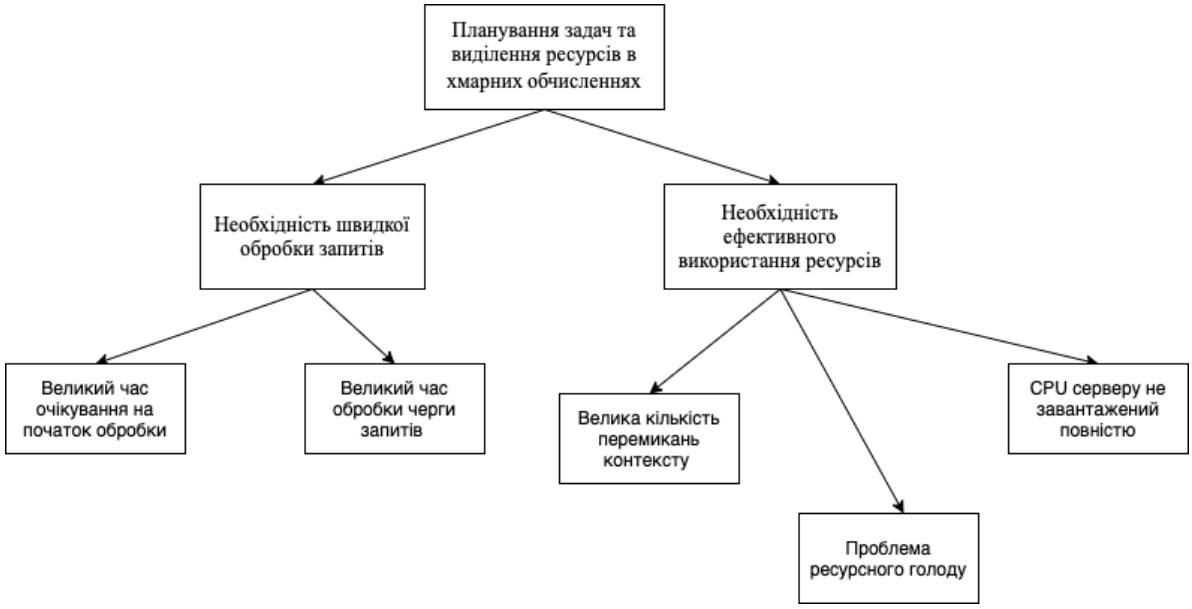


Рис. 40. Дерево проблем

5.2. Зацікавлені сторони

Для ефективної діяльності будь-якої компанії необхідні надійні ресурси. Навіть для збереження та маніпуляції даними необхідно мати сервери для розгортання баз даних. Найбільшою перешкодою розгортання серверів стають високі вхідні вимоги, а саме: якісне, коштовне обладнання, штаб співробітників, які є професіоналами у галузі адміністрування серверів, а також конкуренція у вигляді інших невеликих компаній, які мають більш технологічний операційний стек.

Співпраця з хмарними провайдерами може суттєво спростити багато процесів для будь-якої кампанії: як IT, так і нетехнічних. Міграція своїх технічних засобів до хмарних середовищ суттєво економить кошти, підвищує надійність системи та покращує якість роботи компаній.

Крім цього, в хмарних обчисленнях можуть бути зацікавлені окремі користувачі. Наприклад, наукові дослідники мають потребу у доступі до великих обчислювальних потужностей на короткий час, оскільки дослідження часто обмежені у часі і немає сенсу придбати фізичні сервери. Студенти, наприклад, у сфері машинного навчання, мають потребу доступу до потужностей на період навчання для виконання малих ресурсоємних проєктів. Стартапери зацікавлені у ефективному розподілені коштів на розробку проєкту. Програмісти часто звертаються до хмарних сервісів за їх додатковою функціональністю, яка надається як окремий готовий сервіс.

Потенційними клієнтами проєкту будемо вважати невеликі компанії (аутсорс), що прагнуть користуватися хмарними обчисленнями замість розміщення власного серверу, окремих користувачів, а також державні установи.

Для потенційних клієнтів існує список загальних вимог до сервісу:

- безперебійний цілодобовий доступ;
- висока продуктивність;
- низька вартість;

- цілодобова технічна підтримка.

А також окремі вимоги для державних структур:

- висока надійність;
- захист даних.

Виходячи з вищеописаного, створено таблицю аналізу зацікавлених сторін (табл. 2).

Таблиця 2

Аналіз зацікавлених сторін

Зацікавлена сторона	Інтерес зацікавленої сторони	Вплив зацікавленої сторони	Стратегія приваблення
Компанії (малий та середній бізнес)	Просте використання хмарних ресурсів як альтернатива серверам.	Середній	Розміщення на тематичних виставках. Проведення активної рекламної кампанії. Виступ на тематичних конференціях з доповіддю. Проведення презентацій для представників зацікавлених сторін. Співпраця з університетами.
Компанії (великий бізнес)	Швидке виконання великої кількості задач.	Високий	
Державні установи	Надійність та захист даних.	Високий	
Програмісти	Розширений функціонал хмарних середовищ.	Середній	
Дослідники	Доступ до потужних обчислювальних ресурсів на короткий час.	Середній	

Основна стратегія розвитку проєкту полягає у сильній рекламній компанії на основі стратегії диференційованого маркетингу.

Диференційований маркетинг – це робота на декількох сегментах ринку з пропозицією їм різноманітних товарів і різних наборів засобів формування попиту і стимулювання збуту. Мета такого маркетингу – більш повне освоєння кожного з вибраних секторів і розширення на цій основі збуту і отриманого прибутку.

Використання диференційованого маркетингу призводить до:

- підвищення попиту на продукцію за рахунок її адаптації під вимоги споживачів;
- посилення позиції компанії на ринку;
- збільшення обсягів реалізованих товарів;
- впізнаванню торгової марки.

Диференційований маркетинг спрямований на задоволення потреб споживачів. Компанія, яка прийняла рішення використовувати цю методику, повинна провести оцінку сегментів ринку і вибрати найбільш підходящі для просування своєї продукції або послуги.

5.3. Комерційне рішення. Основні характеристики

Оскільки область хмарних обчислень не дуже молода, тверді лідери вже існують, розберемо найвідоміших.

У наш час в світі є три хмарні гіганти – AWS, Azure, Google Cloud. Ці компанії займають більшу частку ринку всього світу (крім Китаю, там впевнені позиції займає Alibaba Cloud), є технологічними лідерами і задають тренди в розвитку хмарних IaaS сервісів. Наприклад, AWS має в своєму портфоліо більше 100 сервісів.

Проте, можна з впевненістю сказати, що стовідсоткової монополізації ринку вони не досягли. Проаналізуємо, які фактори забезпечують існування інших операторів хмар.

По-перше, бізнес-модель даних гігантів (а саме IaaS) розрахована на чисто хмарні рішення з мінімальними вкрапленнями гібридних інфраструктур. Наявність власних систем віртуалізації (наприклад, модифікованого KVM у AWS) робить процедуру виходу із хмари досить проблематичною.

Дивлячись на варіативність цих сервісів, очевидно, що ціни на деякі з них вище, ніж у інших провайдерів. Це зумовлено тим, кожний з цих провайдерів має величезний набір інструментів. Варто відзначити, що часто завдання не вимагає і 10% із запропонованих сервісів.

Далі йдуть більш дрібні гравці світового ринку, такі як OVH, iLand, GreenCloud. Хоча кожен з них цілком міг би заповнити своїми потужностями український ринок, на даний момент в Україні вони представлені мало. Здебільшого такі провайдери орієнтуються на певний регіон або країну і не поширені за межами свого ареалу. Дані компанії зазвичай використовують одну з комерційних систем віртуалізації (VMWare, Microsoft, OpenStack) і в тій чи іншій мірі позбавлені обмежень гігантів. Розміри таких хмарних IaaS провайдерів можуть варіюватися від великих, що покривають кілька країн, до локальних, діючих в рамках окремої країни.

Настільки низька популярність в Україні пояснюється географічною віддаленістю (латентність і висока вартість побудови каналів), а також складністю підписання контракту, адже для цього потрібна ліцензія на зовнішньоекономічну діяльність.

Одним із прикладів таких локальних сервіс-провайдерів може служити компанія De Novo, орієнтована на ринок України. Безумовно, за кількістю надаваних сервісів українські компанії мають менше шансів вийти на світовий ринок та зрівнятися з гігантами на кшталт AWS, але вони добре розуміють специфіку українського IT та здатні запропонувати більш дешеві послуги.

Таблиця 3

Характеристика потенційного ринку проєкту

№	Показники стану ринку	Характеристика
1	Кількість головних гравців	3
2	Загальний обсяг продаж	–
3	Динаміка ринку	Зростає
4	Наявність обмеження для входу	Високий стартовий капітал, конкуренція
5	Специфічні вимоги до сертифікації	Стандарт ISO/IEC 27001
6	Середня форма рентабельності в галузі	>130%

Виходячи з даної таблиці та думок світових експертів, тенденція хмарних технологій зростає, що визначає економічну вигоду проєкту у цій сфері.

За рахунок великого бізнесу ринок хмарних технологій і послуг з року в рік стрімко нарощує свої обсяги. Спостерігається щорічний приріст на 30–40%. Це швидше за все пов’язано з переходом великих компаній в хмару.

Для великого бізнесу прийнятна так звана гібридна модель, в якій об’єднані: фізичний сервер, приватне хмарне сховище (під критичні бізнес-системи), сховище, запропоноване професійним провайдером (для обробки даних) і хмара на серверах світових компаній (під короткочасні проєкти).

Завдяки інтенсивному залученню інформаційних технологій в більшість бізнес-процесів, потреба компаній в обчислювальних ресурсах тільки зростає. При цьому, масштабування корпоративної IT-інфраструктури з кожним роком компаніям обходиться дорожче.

Тому хмарні послуги сьогодні є дуже зручними для бізнесу. Хмарні сервіси легко масштабувати відносно до потреб бізнесу, це можливість користуватися ресурсами світових вендорів на умовах оренди.

Сьогодні відомі світові вендори пропонують нові варіанти по обробці даних і оптимізації їх зберігання. Також в цьому році продовжується інвестування в робототехніку, штучний інтелект, блокчейн-технології та інші тренди минулого року.

Отже, з огляду на вищесказане, можна дати визначення кінцевого продукту. Це хмарне середовище моделі SaaS, яке реалізовує запропонований у попередній розділах алгоритм виділення ресурсів та алгоритм балансування навантаження. Завдяки описаному способу обробки запитів на виконання задач, сервіс здатний швидше виконувати задачі, займаючи менше фізичних віртуальних машин.

5.4. Конкурентні переваги рішення

Описаний стартап хмарного сервісу має наступні конкурентні переваги:

1. Доступність рішення для малого та середнього бізнесу. Завдяки компактності сервісу, є можливість створити доступні тарифи для користувачів, яким не потрібний доступ до такого широкого спектру сервісів, як наприклад, у AWS.
2. Популярність вибраного напрямку. Люди кожен день користуються хмарними технологіями, навіть того не помічаючи, а компанії все частіше звертаються до хмар, замість того, щоб створювати офісний сервер, адже це в рази вигідніше.
3. Конкурентоздатність. Очевидно, що стартап не може конкурувати з хмарними гігантами, такими як AWS чи Azure, проте можна охопити сегмент ринку, на якому працюють невеликі. Конкурентоздатність можна забезпечити завдяки оптимізації сервісу, його потужностей а також з порівняно низькою ціною на послуги, що завжди привертає увагу користувачів.

Розроблена система має кілька загальних недоліків, які притаманні будь-яким стартапам:

1. Вимагає багато сил та часу, адже програмна реалізація, розробка, а також апаратне налаштування вимагає великих часових вкладень.
2. Значні кошти підуть на розгортання проєкту. Для створення повноцінного хмарного сервісу потрібна команда відповідних фахівців, з високими навичками та поглибленими знаннями у області хмарних технологій. Також, реалізація хостів потребує комплектуючих, що досить затратно. Необхідно забезпечити постійне живлення для роботи хостів, що вимагає певних затрат на електроенергію.
3. Проблема окупності. Як і будь-який стартап, ідея хмарного сервісу хоч і дуже перспективна, існують дуже великі ризики. Для налагодження співробітництва, приваблення клієнтів та впевненого опанування ринку можуть знадобитись роки.

Хоча недоліків у створенні хмарного середовища на даному етапі розробці досить багато, популярність хмарних технологій доводить, що навіть невеликий стартап може досягти успіху, і прикладів такого прориву є багато. Наприклад, ThousandEyes – хмарний сервіс для моніторингу та аналітичної обробки великих обсягів даних. Такі відомі технологічні компанії, як Evernote, Jive, Zynga визнали оригінальність проєкту і якість його обслуговування, почавши співпрацювати з ThousandEyes.

ThousandEyes являється реальним прикладом стартапу. Це перспективний проєкт, який і на даний момент розвивається. Заснували його програмісти з Університету Каліфорнії: Рікардо Олівейра і Мохіто Лід. У 2014 році, після чотирьох років існування проєкту, він різко став набирати популярність. Після того, як почався ряд інвестицій у стартап на чолі з Sequoia Capital, вартість проєкту збільшилась у 4 рази.

Іншим прикладом є стартап Zenefits, що дозволяє власникам бізнесів і HR-менеджерам управляти всіма співробітниками компанії через одну

універсальну платформу.

Заснував Zenefits бізнесмен Паркер Конрад в 2013 році. Всього за рік проєкт з невідомого стартапу перетворився в бізнес, що стрімко розвивається, та оцінюється в приблизно 600 млн доларів. Серед інвесторів компанії – Андерсен Хоровіц і інші провідні фонди Кремнієвої долини. Zenefits зараз обслуговує понад 2000 корпоративних клієнтів.

За допомогою ПЗ від Zenefits менеджери HR-відділів можуть зручніше вести електронну звітність по найму персоналу, заробітної плати, страхування і т. п. Крім того, сервіс забезпечує доступ до системи з будь-якої точки на планеті.

Компанія суттєво порушує стандарти SaaS-бізнесу [14], надаючи всім охочим безкоштовний доступ. Монетизація стартапу заснована на іншому – сервіс заробляє на комісії, надаючи малому бізнесу, зареєстрованому на платформі, свої послуги страхового брокера. Тобто, компанії-клієнти можуть віддати роботу, пов’язану зі страховими виплатами і відрахуваннями, на аутсорс Zenefits. Мабуть, багато клієнтів сервісу користуються такою пропозицією, так як це серйозно спрощує бізнес-процеси. До того ж, якщо весь функціонал HR-відділу вже проходить через Zenefits, то логічно віддати їм на управління і страхування.

Отже, можна зробити висновок, що при вдалій реалізації та залученні інвестицій, можна розробити якісну та конкурентоспроможну систему хварних обчислень.

5.5. Клієнти. Сегменти ринку споживання

Як зазначено у підрозділі 5.2, основними клієнтами вважаються аутсорсингові компанії.

Аутсорсинг (англ. Outsourcing) – це відмова компанії від самостійного виконання ряду некритичних для бізнесу функцій або частин бізнес-процесів і передача їх сторонньому підряднику, який професійно

спеціалізується на наданні таких послуг. Як правило, аутсорсинг відноситься до розряду стратегічних рішень. Головний принцип аутсорсингу – залишити в компанії лише ті процеси, які вона робить краще за інших, а усі інші процеси віддати зовнішнім підрядникам, які також здатні виконати їх краще від усіх.

Головним критерієм для передачі будь-якого бізнес-процесу або бізнес-функції на аутсорсинг є, звичайно, наявність конкурентного середовища. Монополіст рідко є клієнт-орієнтованим і піклується про забезпечення конкурентної ціни на свої послуги.

Основними результатами застосування аутсорсингу є скорочення витрат, відповідно зростає ефективність бізнесу, з'являється можливість вивільнити цілий ряд ресурсів, для розвитку нових напрямків або концентрації на вже існуючих.

Отже, виходячи з основного призначення аутсорсингу – скорочення витрат, очевидно ефективнішим шляхом зберігання даних є оренда хостів у хмарній системі, аніж утримання власних серверів з оплатою електроенергії, найманням технічних спеціалістів для підтримки серверів та без гарантій щодо безперебійної роботи системи.

Для аутсорсингових компаній хмарне сховище даних є вигідним.

На даний момент ситуація у країні щодо хмарних систем не розвинена, що є основним утримуючим фактором для взаємодії з державними замовниками (зріст тенденції присутній).

Виходячи з цього, співпраця з державними структурами на даний момент неможлива.

За даними статистики зросту ринку хмарних обчислень (рис. 41), можна стверджувати, що актуальність розробки ще досить довго буде займати свої позиції, а найбільшим сегментом споживання потенційно буде північно-американський ринок. При цьому в Європі хмарні обчислення також досить популярні і якщо розпочати на ній та залучити потрібні інвестиції, можна опанувати і американский ринок у тому числі.

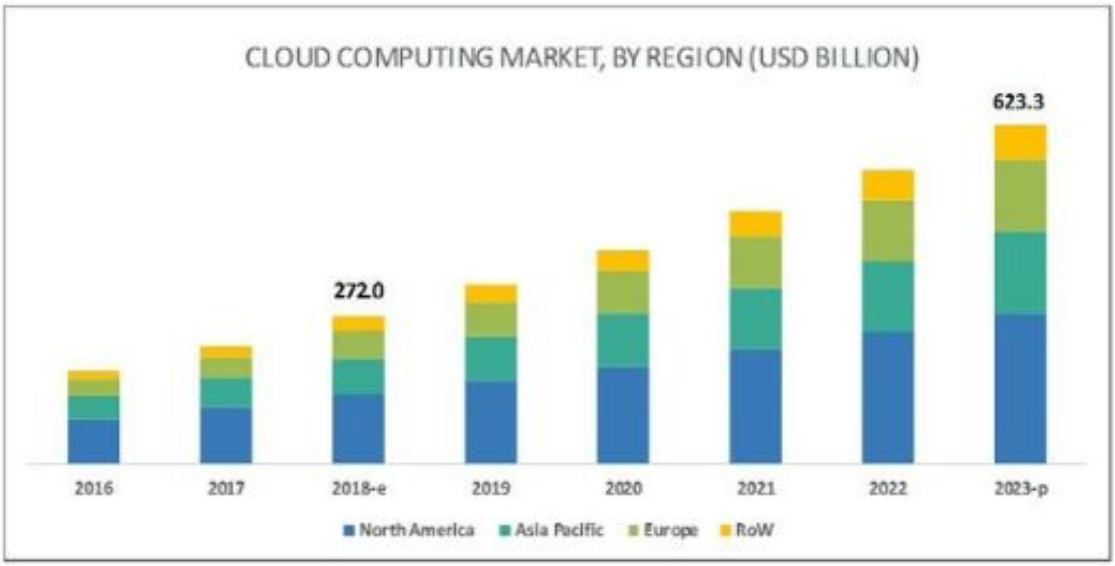


Рис. 41. Статистика зросту ринку хмарних обчислень

5.6. Унікальна ціннісна пропозиція

Враховуючи проблеми користувачів, які були виокремлені у підрозділі 5.1, можна сформулювати унікальну ціннісну пропозицію, яка здатна задовольнити потреби зацікавлених сторін: платформа хмарних обчислень з унікальним методом балансування навантаження та виділення ресурсів, яка має веб-інтерфейс та надає користувачам ресурси за вимогою.

Різниця у ціні послуг сервісів такого ж рівня реалізації та даного сервісу складатиме приблизно 10%. Дана цінова політика знижує роздрібний прибуток, проте за стратегією «дешевше-більше» загальний прибуток повинен перевищувати конкурентний.

Основний напрям у реалізації буде у сторону оптимізації роботи сервісу, що є гарантом якості проєкту. Не менш важливою характеристикою є захищеність баз даних сервісу, що також має реалізуватись у першу чергу.

Прогнозується створення рекламної кампанії з прикладами надання рекомендацій від користувачів сервісу та поширення рекомендацій шляхом розповсюдження інформації користувачами.

5.7. Доходи та витрати

Сумарний дохід складається як сума доходу від оренди хостів на використання програмного забезпечення та надання послуг на їх підтримку.

В хмарному сервісі пропонується послуга оренди хостів для збереження, використання даних та виконання обчислень віддалено. Тарифікація залежить від кількості зайнятої користувачем пам’яті та часу виконання його програмного коду.

Основні витрати на реалізацію сервісу:

1. Заробітня плата команді із 3х розробників: 2 фулстек-розробники та системний адміністратор. Середня ставка – 1500$/місяць.
2. Закупівля серверів Dell PowerEdge R720 з процесором 2 x XEON 6 Core E5-2620 2.0 GHz. Ціна одного серверу – 1000$, для системи необхідно принаймні 20 шт. Витрати – 20 000$.
3. Комутатор CISCO Catalyst 2960-X (WS-C2960X-24TS-LL). Витрати – 1000$.
4. Пристрій безперебійного живлення APC Smart-UPS C 1500VA LCD 230V (SMC1500I), 20 шт. Витрати – 14 000$.
5. Монітор 27" DELL U2719D (210-ARBR) для адміністрування системою – 600$.
6. Оренда приміщення для розміщення серверів – 700$/місяць.
7. Реклама продукту на різних платформах – 1500$/місяць.
8. Споживання електроенергії складатиме приблизно 15000 кВт/міс. Витрати на електроенерію по промисловому тарифу для 2-го класу напруги складає 1200$/міс.
9. Податки.

У розрахунок береться досить дешеве обладнання, що є ще більш вигідним економічно для проєкту.

Вважаючи, що на розробку системи команда з п’яти розробників витратить пів року, розрахуємо потенціальні додатки та витрати у вигляді

таблиці. За цей період витрати на електроенергію будуть набагато менші, оскільки сервери не отримують максимального навантаження. На ринок можна виходити після завершення розробки і тестування. Продаж послуг та рекламні кампанії почнуться з 7-го місяця. З цього моменту можна залишити менше розробників для підтримки системи.

Таблиця 4

Витрати на реалізацію проєкта за перше півріччя

Найменування витрат	1-й міс., т. $	2-й міс., т. $	3-й міс., т. $	4-й міс., т. $	5-й міс., т. $	6-й міс., т. $	Результат за 6 місяців
Витрати							
Зарплатня	4.5	4.5	4.5	4.5	4.5	4.5	27
Оренда приміщення	0.7	0.7	0.7	0.7	0.7	0.7	4.2
Рекламна кампанія	–	–	–	–	–	–	–
Електро-енергія	0.2	0.2	0.2	0.3	0.5	0.5	1.9
Обладнання	35.6	–	–	–	–	–	35.6
Разом витрат	41	5.4	5.4	5.5	5.7	5.7	68.2
Очікувані прибутки							
Продаж послуг	–	–	–	–	–	–	–
Результат без оподаткування							
	–41	–5.4	–5.4	–5.5	–5.7	–5.7	–68.2

Таблиця 5

Витрати на реалізацію проєкта за друге півріччя

Найменування витрат	7-й міс., т. $	8-й міс., т. $	9-й міс., т. $	10-й міс., т. $	11-й міс., т. $	12-й міс., т. $	Результат за півроку
Витрати							
Зарплатня	1	1	1	1	1	1	6
Оренда приміщення	0.7	0.7	0.7	0.7	0.7	0.7	4.2
Рекламна кампанія	1.5	1.5	1.5	1.5	1.5	1.5	9
Електро-енергія	1.2	1.2	1.2	1.2	1.2	1.2	7.2
Обладнання	–	–	–	–	–	–	–
Разом витрат	4.4	4.4	4.4	4.4	4.4	4.4	26.4
Очікувані прибутки							
Продаж послуг	2	3	4	5	7	7	28
Результат без оподаткування							
	–2.4	–1.4	–0.4	0.6	2.6	2.6	1.6

Отже, система буде окупатися досить повільно, приблизно 3 роки, оскільки з вказаною кількістю серверів максимальний можливий прибуток від продажу обчислювальних потужностей становить всього 7000$. Це досить поширена проблема стартапів такого типу. Для того, щоб така розробка мала сенс, необхідно залучати інвесторів та дуже активно масштабуватись, щоб після того, як будуть покриті початкові витрати, прибуток від продаж послуг був досить великим.

5.8. Бізнес-модель

Узагальнимо бізнес-модель.

Споживачі: аутсорсингові компанії, які прагнуть зменшити власні витрати.

Проблеми: неефективне використання обчислювальних ресурсів; складність у налаштуванні та підтримці серверів; для одноразових обчислень придбати сервери не оптимально.

Рішення: програмне забезпечення, що являє собою хмарну систему, яка дає можливість зберігати та керувати даними та автоматично масштабує ресурси, виділені на користувача.

Унікальна ціннісна пропозиція: програмне забезпечення, що реалізує новий метод динамічного виділення ресурсів у хмарних системах; зниження вартості утримування даних для аутсорсингових компаній.

Потоки доходів. Основним типом продажу послуг є надання обчислювальних ресурсів з можливістю сплачувати за мілісекунду їх використання, але не менше ніж 150 мс. Крім цього продається місце в сховищі даних з можливістю сплачувати за кожний мегабайт зайнятого сховища.

Структура витрат: утримання персоналу для надання технічної підтримки; придбання та підтримка апаратної частини; витрати на електроенергію.

Ключові метрики: сумарний час активного навантаження; сумарне зайнятий обсяг сховища даних.

Отже, зважаючи на дані у таблиці 6, можна зробити висновок, що запропонований проєкт, який реалізує описаний у роботі метод динамічного виділення ресурсів у хмарних системах, має перспективи у своїй подальшій реалізації.

Таблиця 6

Канва бізнес-моделі

<table>
<tr><td rowspan="2">Проблема

Неефективне використання обчислювальних ресурсів;
складність у налаштуванні та підтримці серверів;
для одноразових обчислень придбати сервери не оптимально.</td><td>Рішення

Хмарний обчислювальний сервіс з більш оптимальним способом використання ресурсів.</td><td>Унікальна ціннісна пропозиція

Розроблена платформа для хмарних обчислень з веб-інтерфейсом, яка дає можливість економити кошти завдяки покращеному способу балансування навантажень та виділення ресурсів.</td><td rowspan="2">Споживачі

Аутсорсингові компанії;
розробники;
дослідники.</td></tr>
<tr><td>Ключові метрики

Сплачений час розрахунків.
Сплачений обсяг сховища даних.</td><td>Канали

Власна платіжна система у веб-додатку.</td></tr>
<tr><td colspan="2">Структура витрат

Заробітна плата робітникам;
придбання апаратного забезпечення;
витрати на електроенергію та оренду приміщення.</td><td colspan="2">Потоки доходів

Продаж кожної мілісекунди активного навантаження ресурсу (але не менше 150 мс). Продаж кожного Мегабайту сховища даних.</td></tr>
</table>

5.9. Висновки до розділу 5

В цьому розділі був проведений аналіз тенденції хмарних обчислень, виявлено наявні проблеми та підсумовано можливість їх вирішення та вплив на розвиток проєкту. Крім проблем, було виділено основні зацікавлені

сторони у вирішенні їх існуючих потреб, ступінь впливу даних сторін на вирішення проблем. Отже, було запропоновано комерційне рішення з конкурентними перевагами, що задовольняє інтереси потенційних клієнтів, та створена стратегія конкурентоздатності запропонованого продукту.

Проведено аналіз потенційних клієнтів, розгляд ринку споживання. Цей аналіз дав змогу спрогнозувати потенційні доходи та витрати на реалізацію продукту. У результаті була описана бізнес-модель, що обґрунтовує вигоду та сенс реалізації запропонованого продукту та прогнозує його потенційну окупність та прибутковість.

ВИСНОВКИ

Проаналізовано існуючі алгоритми планування задач та розподілення ресурсів CPU у хмарних середовищах, визначено їх основні недоліки, а саме проблеми великого часу обробки задачі, великого часу очікування на початок обробки та голодування ресурсів.

На основі проведеного аналізу запропоновано алгоритмічно-програмний метод планування задач та динамічного виділення віртуальних машин в хмарному середовищі з метою оптимізації використання CPU та пришвидшення виконання задач.

Проведено експериментальні дослідження, які показали, що час обробки завдань при використанні запропонованого гібридного алгоритму на основі алгоритмів Round Robin та Shortest Job Frst зменшується від 10% до 30% (у порівнянні з використанням алгоритмів First-Come–First-Serve та алгоритмом Max-Min відповідно).

Було розроблено метод планування задач та динамічного виділення віртуальних машин, який реалізує запропонований гібридний алгоритм. Систему обробки запитів на виконання задач було розроблено у вигляді консольної утиліти, за допомогою якої користувачі можуть запускати свій програмний код. Обгрунтовано вибір технологій розробки утиліти, описана архітектура системи, а також взаємодія між її сервісами. Було виконано навантажувальне тестування системи та проаналізовано час виконання програмного коду різної складності та для різної кількості запитів.

Проведено огляд ринку хмарних середовищ, виокремлено основні зацікавлені сторони та їх проблеми. Розглянуто основних конкурентів та сформована унікальна ціннісна пропозиція, яка спряє конкурентоспроможності реалізованого проєкту.

У майбутньому було б доцільно розробити веб-інтерфейс для зручного користування системою з вбудованим IDE. Також було б доцільно розробити окремі сервіси, якими клієнти могли б користуватись ізольовано

від інших.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. U. Pawde, A. Ingole, S. Chavan. An Optimized Algorithm for Task Scheduling based on Activity based Costing in Cloud Computing(2011).
2. Michael J. Kavis Architecting the Cloud: Design Decisions for Cloud Computing Service Models (SaaS, PaaS, and IaaS).
3. T. Erl, R. Puttini, Z. Mahmood Cloud Computing: Concepts, Technology & Architecture.
4. Mell, P. The NIST Definition of Cloud Computing [Electronic resource] / Peter Mell, Timothy Grance. – Recommendations of the National Institute of Standards and Technology NIST. – 2011. – SP 800-145. – Електронні дані. –Доступ: https://csrc.nist.gov/publications/detail/sp/800-145/final.
5. Bass L. Software Architecture In Practice, Second Edition / L. Bass, P. Clements, R. Kazman. – Boston: Addison-Wesley. – 2003. – ISBN 0-321-15495-9.
6. Donaldson V. Program Speedup in a Heterogeneous Computing Network / V. Donaldson, F. Berman, R. Paturi. // Journal of Parallel and Distributed Computing. – September 1994. – Vol. 21. – No 3.
7. Leopold C. Parallel and Distributed Computing. A survey of Models, Paradigms, and Approaches Wiley Series on Parallel and Distributed Computing / C. Leopold // Albert Zomaya Series Editor. – 2001.
8. Cloud computing: distributed internet computing for IT and scientific research / [M.D. Dikaiakos, D. Katsaros, P. Mehra and others.] . – Internet Computing, IEEE. – 2009. – №13.
9. Rima B.A Taxonomy and Survey of Cloud Computing Systems / B.P. Rima, E. Choi, and I. Lumb // Proceedings of 5th IEEE International Joint Conference on INC, IMS and IDC, Seoul. – Korea, August 2009.
10. Alakeel A.A Guide to dynamic Load balancing in Distributed Computer Systems" / A. M. Alakeel // International Journal of Computer Science and Network Security (IJCSNS) . – Vol. 10. – No. 6. – June 2010.

11. Mykel J. Kochenderfer and Tim A. Wheeler. 2019. Algorithms for Optimization. The MIT Press.
12. Korte B, Vygen J (2018) Linear programming algorithms. Combinatorial optimization. Springer, Berlin, pp 75–102.
13. George Reese, Cloud Application Architectures. 2009. O'Reilly Media, Inc. ISBN: 9780596156367.
14. IaaS для бизнеса по кирпичикам [Електронний ресурс]– Режим доступу: https://www.it-grad.ru/files/ebooks/IaaS_po_kirpichikam.pdf.
15. Фундаментальная теория облачных вычислений. А.Е. Кононюк. – [Електронний ресурс] – Режим доступу: http://ecat.diit.edu.ua/ft/CloudTech_1.pdf.
16. Zhong Xu. Performance Study of Load Balanacing Algorithms in Distributed Web Server Systems / Zhong Xu, Rong Huang // CS213 Parallel and Distributed Processing Project Report. 2009.
17. P.Warstein/ Load balancing in a cluster computer / P.Warstein, H.Situ and Z.Huang // In proceeding of the seventh International Conference on Parallel and Distributed Computing, Applications and Technologies, IEEE. – 2010.
18. Tziritas N. On minimizing the resource consumption of cloud applications using process migrations / N. Tziritas // Journal of Parallel and Distributed Computing. – 2013.
19. Load analysis and structural consideration [Електронний ресурс]– Режим доступу: https://www.slideshare.net/kuashaknight/load-analysis-and-structuralconsideration.
20. Kumar M. Dynamic load balancing algorithm for balancing the workload among virtual machine in cloud computing / M. Kumar, S.C. Sharmab – 2017.
21. Kumar, Ranjan & Sahoo, Gadadhar. (2014). Cloud Computing Simulation Using CloudSim. International Journal of Engineering Trends and Technology. 8. 10.14445/22315381/IJETT-V8P216.

22. Himani, Himani & Sidhu, Harmanbir. (2014). Comparative Analysis of Scheduling Algorithms of Cloudsim in Cloud Computing. International Journal of Computer Applications. 97. 29-33. 10.5120/17092-7629.
23. Calheiros, Rodrigo & Ranjan, R. & De Rose, Cesar & Buyya, Rajkumar. (2009). CloudSim: A Novel Framework for Modeling and Simulation of Cloud Computing Infrastructures and Services.

ДОДАТКИ

Додаток 1

Копії графічних матеріалів

Блок-схема запропонованого алгоритму

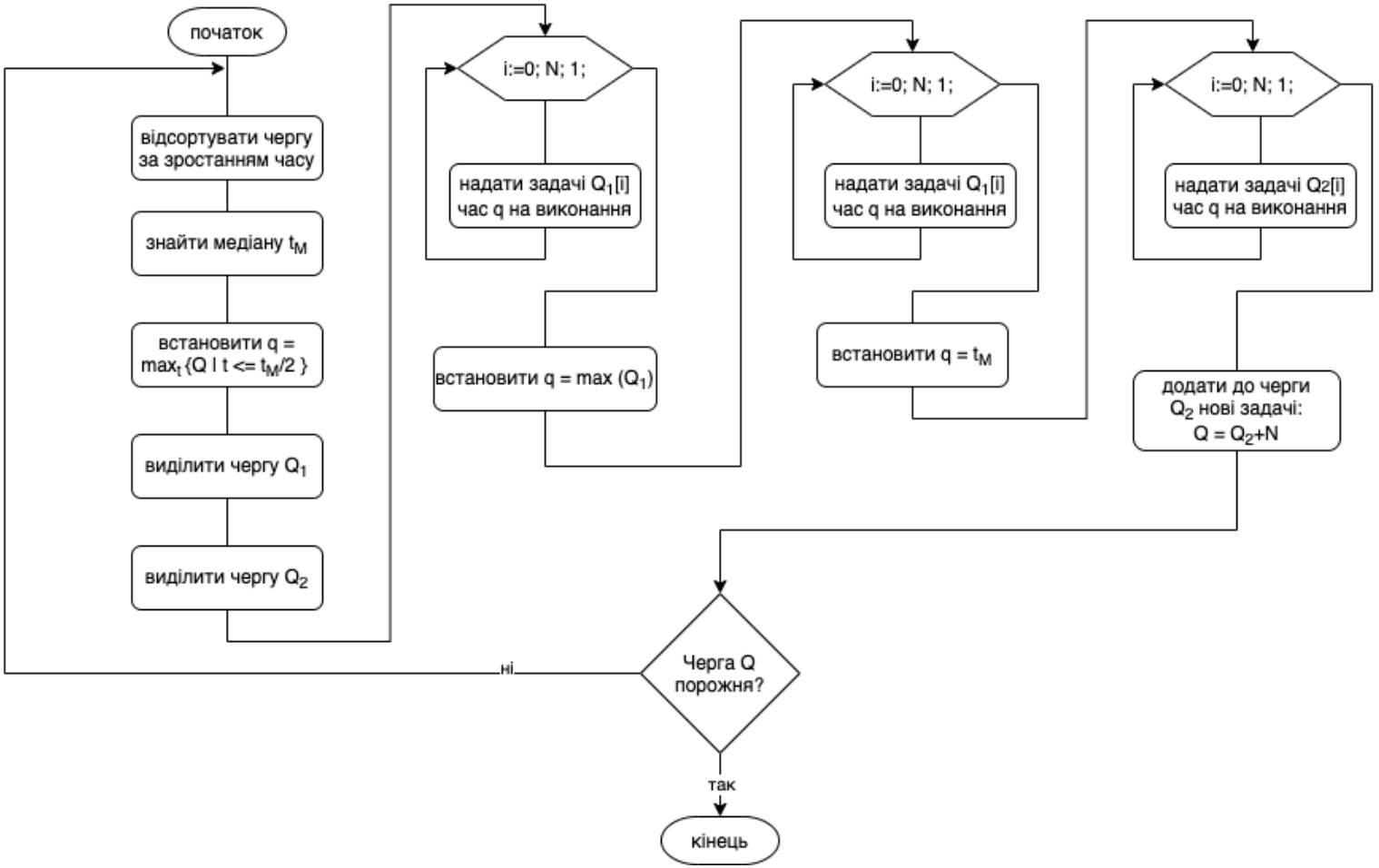


Схема структури системи

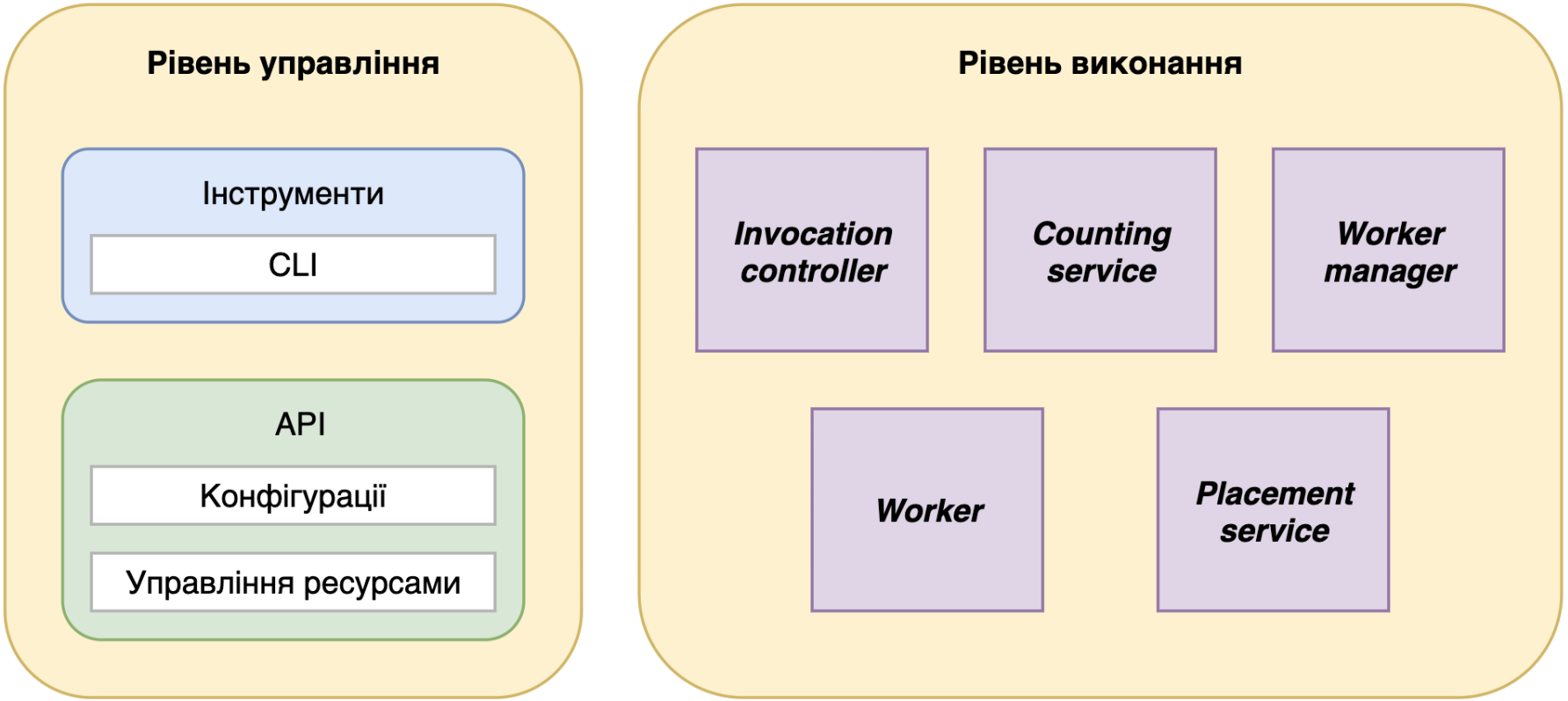
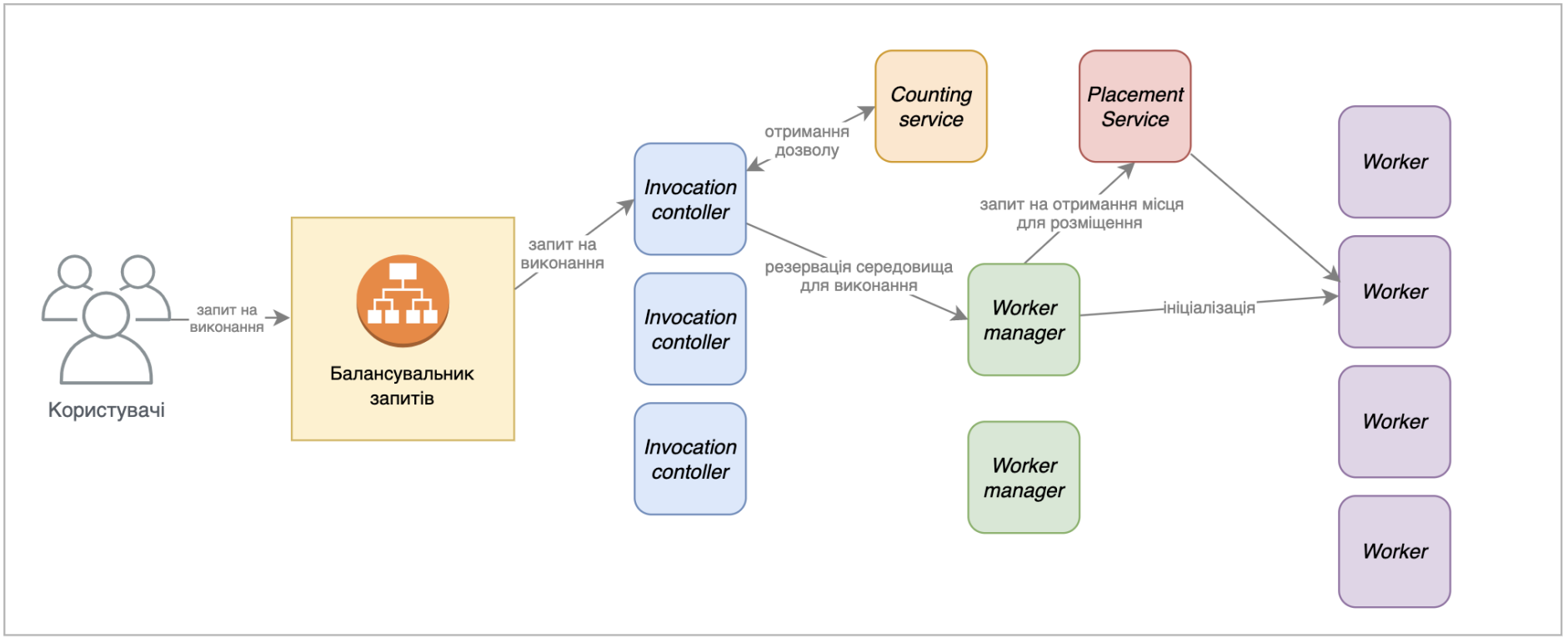
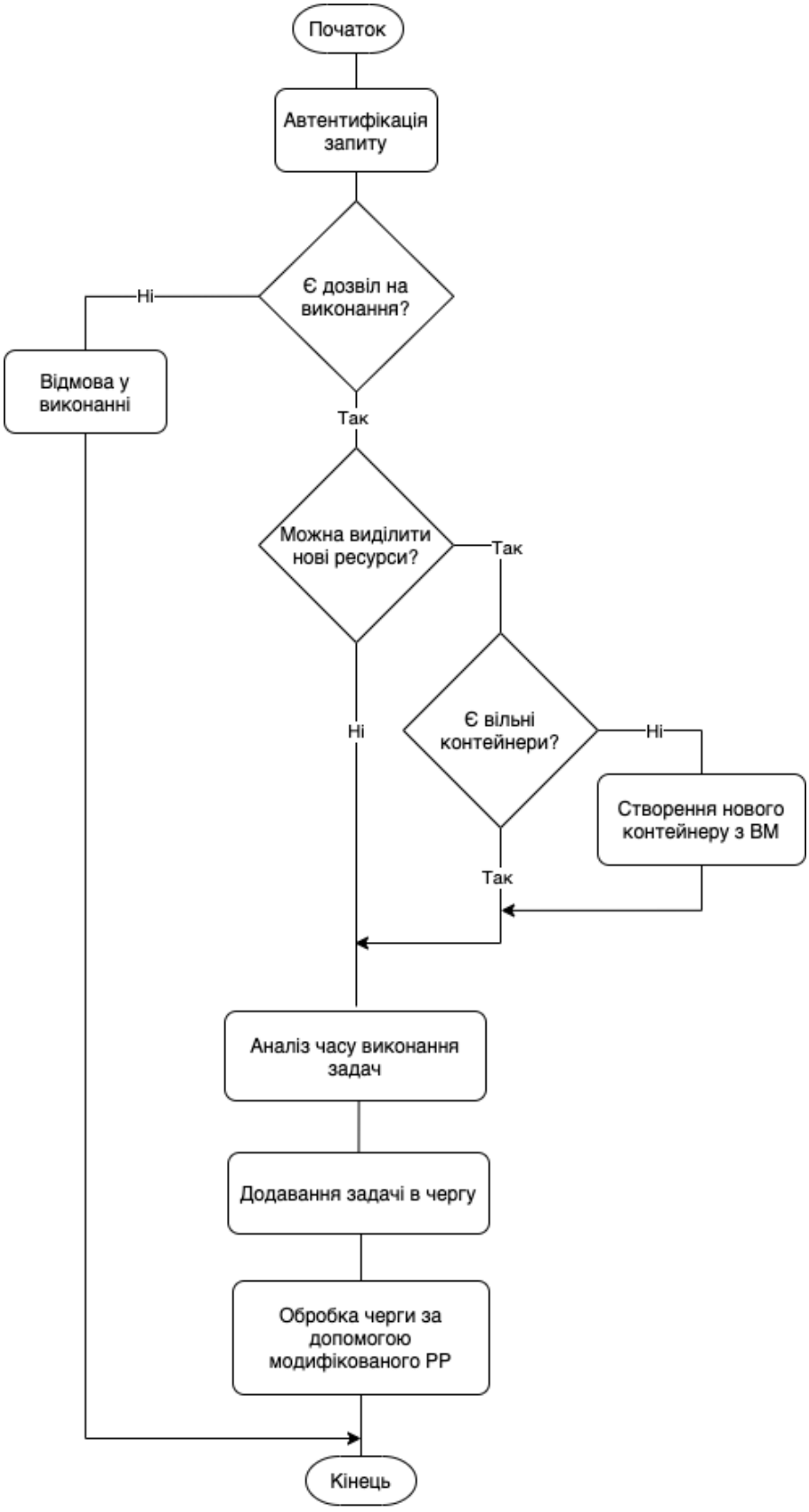


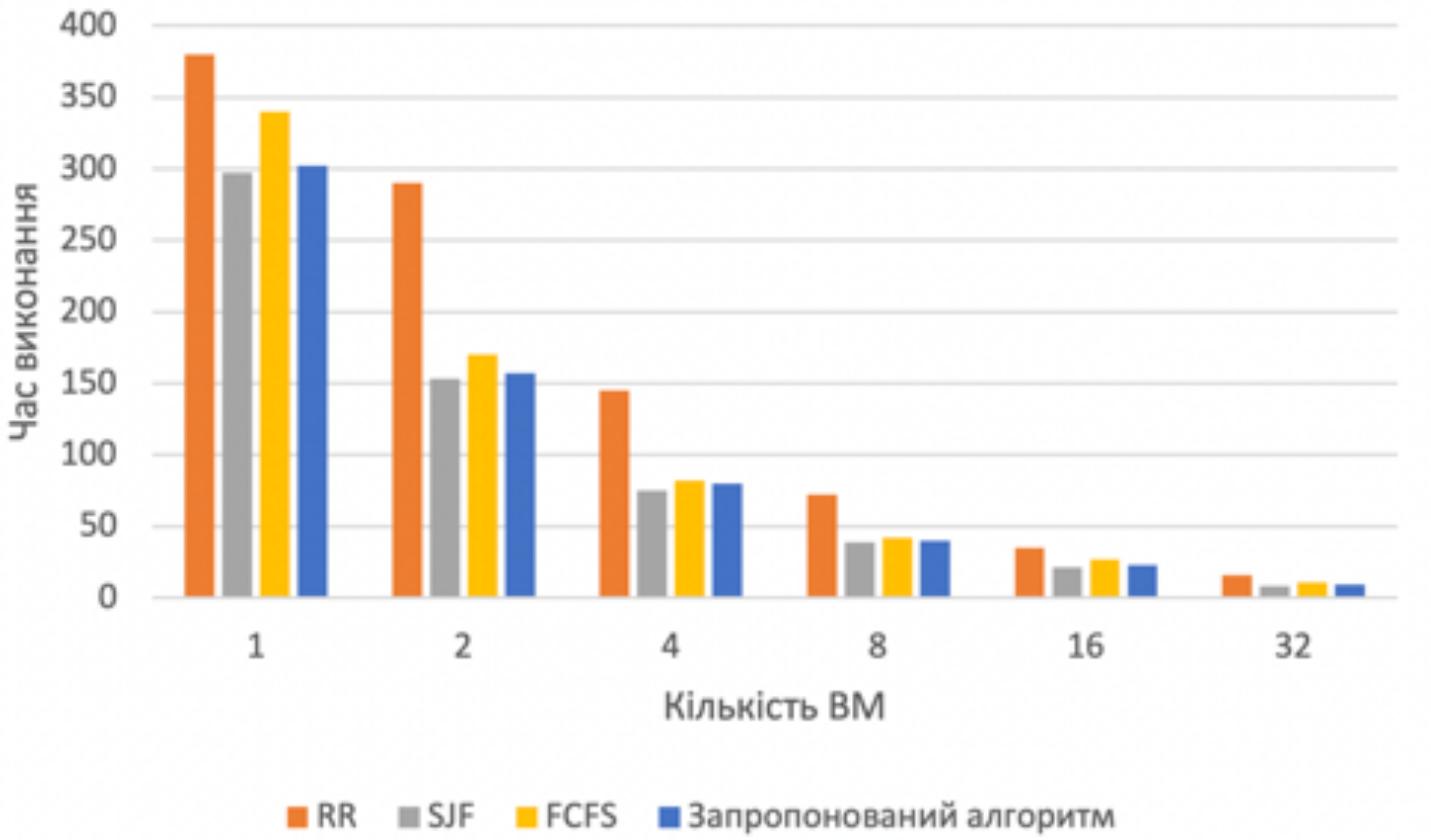
Схема взаємодії сервісів системи



Блок схема обробки запиту



Середній час виконання 1000 запитів для різної кількості віртуальних машин



Канва бізнес-моделі

<table>
<tr><td>Проблема

Неефективне використання обчислювальних ресурсів;
складність у налаштуванні та підтримці серверів;
для одноразових обчислень придбати сервери не оптимально.</td><td>Рішення

Хмарний обчислювальний сервіс з більш оптимальним способом використання ресурсів.</td><td>Унікальна ціннісна пропозиція

Розроблена платформа для хмарних обчислень з веб-інтерфейсом, яка дає можливість економити кошти завдяки покращеному способу балансування навантажень та виділення ресурсів.</td><td rowspan="2">Споживачі

Аутсорсингові компанії;
розробники;
дослідники.</td></tr>
<tr><td colspan="2">Ключові метрики

Сплачений час розрахунків.
Сплачений обсяг сховища даних.</td><td>Канали

Власна платіжна система у веб-додатку.</td></tr>
<tr><td colspan="2">Структура витрат

Заробітна плата робітникам;
придбання апаратного забезпечення;
витрати на електроенергію та оренду приміщення.</td><td colspan="2">Потоки доходів

Продаж кожної мілісекунди активного навантаження ресурсу (але не менше 150 мс). Продаж кожного Мегабайту сховища даних.</td></tr>
</table>

Додаток 2

Лістинг програми

Лістинг 1. worker/client.js

```
const grpc = require("grpc");
const protoLoader = require("@grpc/proto-loader");



const BOOKS_SERVICE_PATH = process.env.BOOKS_SERVICE_PATH || "0.0.0.0:9080";
const PROTO_FILE = `${__dirname}/proto/worker.proto`;

const packageDefinition = protoLoader.loadSync(PROTO_FILE);
const protoDescriptor = grpc.loadPackageDefinition(packageDefinition);
const Client = protoDescriptor.worker.WorkerService;



const client = new Client(
  BOOKS_SERVICE_PATH,
  grpc.credentials.createInsecure()
);

client.createVM({}, (err, res) => {
  console.log('RES');
});
```

Лістинг 2. worker/rpc-server.js

```
const grpc = require("grpc");
const WorkerService = require('./proto/workerService');
const WorkerInterface = require('./proto/workerInterface')();


const PORT = process.env.PORT || "9080";

const getServer = () => {
  const server = new grpc.Server();
  server.addService(WorkerInterface, WorkerService);
  return server;
}

const workerServer = getServer();
workerServer.bind(`0.0.0.0:${PORT}`, grpc.ServerCredentials.createInsecure());
workerServer.start();
console.log(`Server running on port ${PORT}`);
```

Лістинг 3. worker-manager/counter.js

```
const counterObj = {};

const DEFAULT_LIMIT = 4;

module.exports = {
  getConcurrency: (templateId) => {
    const concurrencyValue = counterObj[templateId] || 0;
    return { scale: concurrencyValue < DEFAULT_LIMIT, count: concurrencyValue }
  },
  updateConcurrency: (templateId) => {
    const prevValue = counterObj[templateId] || 0;
    counterObj[templateId] = prevValue + 1;
  },
}
```

Лістинг 4. worker-manager/process.js

```
const { v4: uuidv4 } = require('uuid');
const WorkerRPCClient = require('./worker-rpc-client');

function quickselect(arr, k, getFn = () => {}) {
  if (arr.length == 1) return arr[0];
  else {
    const pivot = arr[0];
    const lows = arr.filter((e) => getFn(e) < pivot);
    const highs = arr.filter((e) => getFn(e) > pivot);
    const pivots = arr.filter((e) => getFn(e) == pivot);
    if (k < lows.length) return quickselect(lows, k, getFn);
    else if (k < lows.length + pivots.length) return pivot;
    else return quickselect(highs, k - lows.length - pivots.length, getFn);
  }
}

function quickselectMedian(arr, getFn) {
  const L = arr.length,
    halfL = L / 2;
  if (L % 2 == 1) return quickselect(arr, halfL, getFn);
  else return 0.5 * (quickselect(arr, halfL - 1), getFn + quickselect(arr, halfL,
getFn));
}

async function startTaskProcessing(task) {
  if (task.isFinished) return true;
  if(task.vmId) {
    await WorkerRPCClient.restoreFromSnapshot({ vmId });
  } else {
```

```
    const { vmId } = await WorkerRPCClient.createVM({ templateId: task.templateId,
accountId: 'default', requestId: uuidv4() });
    task.vmId = vmId;
  }
  return false;
}

async function iterationWithTimeSlice([task, ...others], timeSlice) {
  if(!task) return Promise.resolve();
  return startTaskProcessing(task).then((done) => {
    if(!done) {
      return new Promise(( resolve, reject) => {
        setTimeout(async () => {
          await WorkerRPCClient.makeSnapshot({ vmId: task.vmId  });
          task.eet -= timeSlice;
          resolve(iterationWithTimeSlice(others, timeSlice));
        }, timeSlice)
      });
    }
  })
};

async function processResouceGroup(tasks) {
  const sortedTasks = tasks.sort((a, b) => a - b);
  const median = quickselectMedian(sortedTasks, (task) => task.eet);
  const q1 = tasks.reduce((acc, task) => {
    if (task.eet >= median / 2) return acc;
    return acc > task.eet ? acc : task.eet;
  }, tasks[0].eet);

  const queuesObj = tasks.reduce((acc, task) => {
    if(task.eet > 2 * q1) acc.low.push(task);
    else acc.high.push(task);
    return acc;
  }, { high: [], low: [] });

  const highI1 = await iterationWithTimeSlice(queuesObj.high, q1);
  const maxTimeSliceLeft = Math.max.apply(Math, highI1.map(function(t) { return
t.eet; }))
  const highI2 = await iterationWithTimeSlice(highI1, maxTimeSliceLeft);
  const lowI1 = await iterationWithTimeSlice(queuesObj.low, median);
  return [...highI2,...lowI1];
}

async function run() {
  while(true) {
    const tasks = getTasks(); //TODO:
    const tasksObj = tasks.reduce((acc, task) => {
      if (!acc[task.accountId]) acc[task.accountId] = [];
      acc[task.templateId].push(task);
      return acc;
```

```
    }, {});

    await Promise.all(Object.keys(tasksObj).map(key =>
processResouceGroup(tasksObj[key])));
  }
}

run();
```

Лістинг 5. worker-manager/rpc-server.js

```
const grpc = require("grpc");
const WorkerManagerService = require('./proto/workerManagerService');
const WorkerManagerInterface = require('./proto/workerManagerInterface')();


const PORT = process.env.PORT || "9090";

const getServer = () => {
  const server = new grpc.Server();
  server.addService(WorkerManagerInterface, WorkerManagerService);
  return server;
}

const workerManagerServer = getServer();
workerManagerServer.bind(`0.0.0.0:${PORT}`,
grpc.ServerCredentials.createInsecure());
workerManagerServer.start();
console.log(`Server running on port ${PORT}`);
```

Лістинг 6. worker-manager/worker-rpc-client.js

```
const grpc = require("grpc");
const protoLoader = require("@grpc/proto-loader");



const BOOKS_SERVICE_PATH = process.env.BOOKS_SERVICE_PATH || "0.0.0.0:9080";
const PROTO_FILE = `${__dirname}/../worker/proto/worker.proto`;

const packageDefinition = protoLoader.loadSync(PROTO_FILE);
const protoDescriptor = grpc.loadPackageDefinition(packageDefinition);
const Client = protoDescriptor.worker.WorkerService;



const client = new Client(
  BOOKS_SERVICE_PATH,
```

```
  grpc.credentials.createInsecure()
);

module.exports = client;
```

Лістинг 7. Worker/firecracker/drive.js

```
var Drive = function (modem, drive_id) {
  this.id = drive_id;
  this.modem = modem;
};

Drive.prototype.updatePreboot = function (data, callback) {
  var self = this;
  if (!callback && typeof opts === "function") {
    callback = opts;
    opts = undefined;
  }

  data.drive_id = this.id;

  var optsf = {
    path: "/drives/" + this.id,
    method: "PUT",
    data: data,
    statusCodes: {
      204: true,
      400: "Drive cannot be created/updated due to bad input",
    },
  };

  if (callback === undefined) {
    return new Promise(function (resolve, reject) {
      self.modem.dial(optsf, function (err, data) {
        if (err) {
          return reject(err);
        }
        resolve(data);
      });
    });
  } else {
    this.modem.dial(optsf, function (err, data) {
      callback(err, data);
    });
  }
};
```

```
Drive.prototype.createPreboot = function (data, callback) {
  return this.updatePreboot.apply(this, [data, callback]);
};

Drive.prototype.updatePostboot = function (data, callback) {
  var self = this;
  if (!callback && typeof opts === "function") {
    callback = opts;
    opts = undefined;
  }

  data.drive_id = this.id;

  var optsf = {
    path: "/drives/" + this.id,
    method: "PATCH",
    data: data,
    statusCodes: {
      204: true,
      400: "Drive cannot be updated due to bad input",
    },
  };

  if (callback === undefined) {
    return new Promise(function (resolve, reject) {
      self.modem.dial(optsf, function (err, data) {
        if (err) {
          return reject(err);
        }
        resolve(data);
      });
    });
  } else {
    this.modem.dial(optsf, function (err, data) {
      callback(err, data);
    });
  }
};

module.exports = Drive;
```

Лістинг 8. worker/firecracker.index.js

```
const http = require('http'),
  https = require('https'),
  fs = require('fs'),
  path = require('path'),
  child_process = require('child_process');

const Modem = require('./modem'),
```

```
  Drive = require('./drive'),
  Interface = require('./interface'),
  MMDS = require('./mmds'),
  MachineConfig = require('./vm-config');

var Firecracker = function (opts) {
  this.options = opts;
  this.modem = new Modem(opts);
};

Firecracker.prototype.info = function (callback) {
  var self = this;
  if (!callback && typeof opts === 'function') {
    callback = opts;
    opts = undefined;
  }

  var optsf = {
    path: '/',
    method: 'GET',
    statusCodes: {
      200: true
    }
  };

  if (callback === undefined) {
    return new Promise(function (resolve, reject) {
      self.modem.dial(optsf, function (err, data) {
        if (err) {
          return reject(err);
        }
        resolve(data);
      });
    });
  } else {
    this.modem.dial(optsf, function (err, data) {
      callback(err, data);
    });
  }
};

Firecracker.prototype.action = function (action, callback) {
  var self = this;
  if (!callback && typeof opts === 'function') {
    callback = opts;
    opts = undefined;
  }

  var optsf = {
    path: '/actions',
    method: 'PUT',
```

```
    data: { 'action_type': action },
    statusCodes: {
      204: true,
      400: 'The action cannot be executed due to bad input'
    }
  };

  if (callback === undefined) {
    return new Promise(function (resolve, reject) {
      self.modem.dial(optsf, function (err, data) {
        if (err) {
          return reject(err);
        }
        resolve(data);
      });
    });
  } else {
    this.modem.dial(optsf, function (err, data) {
      callback(err, data);
    });
  }
};

Firecracker.prototype.pause = function (callback) {
  var self = this;
  if (!callback && typeof opts === 'function') {
    callback = opts;
    opts = undefined;
  }

  var optsf = {
    path: '/vm',
    method: 'PATCH',
    data: {
      state: "Paused"
    },
  };

  if (callback === undefined) {
    return new Promise(function (resolve, reject) {
      self.modem.dial(optsf, function (err, data) {
        if (err) {
          return reject(err);
        }
        resolve(data);
      });
    });
  } else {
    this.modem.dial(optsf, function (err, data) {
      callback(err, data);
    });
```

```
  }
};

Firecracker.prototype.resume = function (callback) {
  var self = this;
  if (!callback && typeof opts === 'function') {
    callback = opts;
    opts = undefined;
  }

  var optsf = {
    path: '/vm',
    method: 'PATCH',
    data: {
      state: "Resumed"
    },
  };

  if (callback === undefined) {
    return new Promise(function (resolve, reject) {
      self.modem.dial(optsf, function (err, data) {
        if (err) {
          return reject(err);
        }
        resolve(data);
      });
    });
  } else {
    this.modem.dial(optsf, function (err, data) {
      callback(err, data);
    });
  }
};

Firecracker.prototype.createSnapshot = function (data,callback) {
  var self = this;
  if (!callback && typeof opts === 'function') {
    callback = opts;
    opts = undefined;
  }

  var optsf = {
    path: '/snapshot/create',
    method: 'PUT',
    data,
  };

  if (callback === undefined) {
    return new Promise(function (resolve, reject) {
      self.modem.dial(optsf, function (err, data) {
        if (err) {
```

```
          return reject(err);
        }
        resolve(data);
      });
    });
  } else {
    this.modem.dial(optsf, function (err, data) {
      callback(err, data);
    });
  }
};

Firecracker.prototype.loadSnapshot = function (data,callback) {
  var self = this;
  if (!callback && typeof opts === 'function') {
    callback = opts;
    opts = undefined;
  }

  var optsf = {
    path: '/snapshot/load',
    method: 'PUT',
    data,
  };

  if (callback === undefined) {
    return new Promise(function (resolve, reject) {
      self.modem.dial(optsf, function (err, data) {
        if (err) {
          return reject(err);
        }
        resolve(data);
      });
    });
  } else {
    this.modem.dial(optsf, function (err, data) {
      callback(err, data);
    });
  }
};



Firecracker.prototype.bootSource = function (data, callback) {
  var self = this;
  if (!callback && typeof opts === 'function') {
    callback = opts;
    opts = undefined;
  }

  var optsf = {
```

```
    path: '/boot-source',
    method: 'PUT',
    data: data,
    statusCodes: {
      204: true,
      400: 'Boot source cannot be created due to bad input'
    }
  };

  if (callback === undefined) {
    return new Promise(function (resolve, reject) {
      self.modem.dial(optsf, function (err, data) {
        if (err) {
          return reject(err);
        }
        resolve(data);
      });
    });
  } else {
    this.modem.dial(optsf, function (err, data) {
      callback(err, data);
    });
  }
};

Firecracker.prototype.mmds = function () {
  return new MMDS(this.modem);
};

Firecracker.prototype.drive = function (id) {
  return new Drive(this.modem, id);
};

Firecracker.prototype.interface = function (id) {
  return new Interface(this.modem, id);
};

Firecracker.prototype.machineConfig = function () {
  return new MachineConfig(this.modem);
};

Firecracker.prototype.logger = function (data, callback) {
  var self = this;
  if (!callback && typeof opts === 'function') {
    callback = opts;
    opts = undefined;
  }

  var optsf = {
    path: '/logger',
    method: 'PUT',
```

```
    data: data,
    statusCodes: {
      204: true,
      400: 'Logger cannot be initialized due to bad input.'
    }
  };

  if (callback === undefined) {
    return new Promise(function (resolve, reject) {
      self.modem.dial(optsf, function (err, data) {
        if (err) {
          return reject(err);
        }
        resolve(data);
      });
    });
  } else {
    this.modem.dial(optsf, function (err, data) {
      callback(err, data);
    });
  }
};

Firecracker.prototype.metrics = function (data, callback) {
  var self = this;
  if (!callback && typeof opts === 'function') {
    callback = opts;
    opts = undefined;
  }

  var optsf = {
    path: '/metrics',
    method: 'PUT',
    data: data,
    statusCodes: {
      204: true,
      400: 'Metrics system cannot be initialized due to bad input.'
    }
  };

  if (callback === undefined) {
    return new Promise(function (resolve, reject) {
      self.modem.dial(optsf, function (err, data) {
        if (err) {
          return reject(err);
        }
        resolve(data);
      });
    });
  } else {
    this.modem.dial(optsf, function (err, data) {
```

```
      callback(err, data);
    });
  }
};

Firecracker.prototype.vsock = function (data, callback) {
  var self = this;
  if (!callback && typeof opts === 'function') {
    callback = opts;
    opts = undefined;
  }

  var optsf = {
    path: '/vsock',
    method: 'PUT',
    data: data,
    statusCodes: {
      204: true,
      400: 'Vsock cannot be created due to bad input'
    }
  };

  if (callback === undefined) {
    return new Promise(function (resolve, reject) {
      self.modem.dial(optsf, function (err, data) {
        if (err) {
          return reject(err);
        }
        resolve(data);
      });
    });
  } else {
    this.modem.dial(optsf, function (err, data) {
      callback(err, data);
    });
  }
};

Firecracker.prototype.downloadImage = function (url, dest) {
  const uri = new URL(url);
  var proto = http;
  if (uri.protocol === 'https:') {
    proto = https;
  }

  //var filename = path.posix.basename(uri.pathname);
  //dest = dest + path.delimiter + filename;

  return new Promise((resolve, reject) => {
    if (fs.existsSync(dest)) {
      return reject('File already exists');
```

```
    }

    let file = fs.createWriteStream(dest, { flags: 'wx' });

    file.on('finish', () => {
      resolve();
    });

    file.on('error', err => {
      file.close();
      fs.unlink(dest, () => { });
      reject(err.message);
    });

    const request = proto.get(url, response => {
      if (response.statusCode === 200) {
        response.pipe(file);
      } else {
        file.close();
        fs.unlink(dest, () => { });
        reject(`Server responded with ${response.statusCode}:
${response.statusMessage}`);
      }
    });

    request.on('error', err => {
      file.close();
      fs.unlink(dest, () => { });
      reject(err.message);
    });
  });
};

Firecracker.prototype.spawn = function (binPath) {
  var self = this;
  binPath = binPath || '/usr/bin/firecracker';

  return new Promise(function (resolve, reject) {
    self.child = child_process.spawn(binPath, ['--api-sock',
self.options.socketPath], { detached: true });

    self.child.on('exit', function (code, signal) {
      fs.unlink(self.options.socketPath, () => { });
      self.child = undefined;
    });

    self.child.on('close', function (code, signal) {
      fs.unlink(self.options.socketPath, () => { });
      self.child = undefined;
    });
```

```
    self.child.on('error', function (err) {
      fs.unlink(self.options.socketPath, () => { });
    });

    resolve(self.child);
  });
};

Firecracker.prototype.kill = function () {
  var killed = this.child.kill();
  if(killed === true) {
    fs.unlink(this.options.socketPath, () => { });
    this.child = undefined;
  }
  return killed;
};

module.exports = Firecracker;
```

Лістинг 9. worker/firecracker.interface.js

```
var Interface = function (modem, interface_id) {
  this.id = interface_id;
  this.modem = modem;
};

Interface.prototype.create = function (data, callback) {
  var self = this;
  if (!callback && typeof opts === "function") {
    callback = opts;
    opts = undefined;
  }

  data.iface_id = this.id;

  var optsf = {
    path: "/network-interfaces/" + this.id,
    method: "PUT",
    data: data,
    statusCodes: {
      204: true,
      400: "Network interface cannot be created due to bad input",
    },
  };

  if (callback === undefined) {
    return new Promise(function (resolve, reject) {
      self.modem.dial(optsf, function (err, data) {
        if (err) {
          return reject(err);
```

```
        }
        resolve(data);
      });
    });
  } else {
    this.modem.dial(optsf, function (err, data) {
      callback(err, data);
    });
  }
};

Interface.prototype.update = function (data, callback) {
  var self = this;
  if (!callback && typeof opts === "function") {
    callback = opts;
    opts = undefined;
  }

  data.iface_id = this.id;

  var optsf = {
    path: "/network-interfaces/" + this.id,
    method: "PATCH",
    data: data,
    statusCodes: {
      204: true,
      400: "Network interface cannot be updated due to bad input",
    },
  };

  if (callback === undefined) {
    return new Promise(function (resolve, reject) {
      self.modem.dial(optsf, function (err, data) {
        if (err) {
          return reject(err);
        }
        resolve(data);
      });
    });
  } else {
    this.modem.dial(optsf, function (err, data) {
      callback(err, data);
    });
  }
};

module.exports = Interface;
```

Лістинг 10. worker/firecracker.mmds.js

```
var MMDS = function(modem) {
  this.modem = modem;
};


MMDS.prototype.create = function (callback) {
  var self = this;
  if (!callback && typeof opts === 'function') {
    callback = opts;
    opts = undefined;
  }

  var optsf = {
    path: '/mmds',
    method: 'PUT',
    data: {},
    statusCodes: {
      204: true,
      400: 'MMDS data store cannot be created due to bad input.'
    }
  };

  if (callback === undefined) {
    return new Promise(function (resolve, reject) {
      self.modem.dial(optsf, function (err, data) {
        if (err) {
          return reject(err);
        }
        resolve(data);
      });
    });
  } else {
    this.modem.dial(optsf, function (err, data) {
      callback(err, data);
    });
  }
};

MMDS.prototype.get = function (callback) {
  var self = this;
  if (!callback && typeof opts === 'function') {
    callback = opts;
    opts = undefined;
  }

  var optsf = {
    path: '/mmds',
    method: 'GET',
    statusCodes: {
      204: true,
```

```
      400: 'Cannot get the MMDS data store due to bad input.'
    }
  };

  if (callback === undefined) {
    return new Promise(function (resolve, reject) {
      self.modem.dial(optsf, function (err, data) {
        if (err) {
          return reject(err);
        }
        resolve(data);
      });
    });
  } else {
    this.modem.dial(optsf, function (err, data) {
      callback(err, data);
    });
  }
};

MMDS.prototype.update = function (data, callback) {
  var self = this;
  if (!callback && typeof opts === 'function') {
    callback = opts;
    opts = undefined;
  }

  var optsf = {
    path: '/mmds/',
    method: 'PATCH',
    data: data,
    statusCodes: {
      204: true,
      400: 'MMDS data store cannot be updated due to bad input.'
    }
  };

  if (callback === undefined) {
    return new Promise(function (resolve, reject) {
      self.modem.dial(optsf, function (err, data) {
        if (err) {
          return reject(err);
        }
        resolve(data);
      });
    });
  } else {
    this.modem.dial(optsf, function (err, data) {
      callback(err, data);
    });
  }
```

```
};

module.exports = MMDS;
```

Лістинг 11. worker/firecracker.mmds.js

```
var http = require('http'),
  debug = require('debug')('modem'),
  util = require('util');


var Modem = function (options) {
  this.socketPath = options.socketPath;
  this.timeout = options.timeout;
  this.connectionTimeout = options.connectionTimeout;
  this.headers = options.headers || {};
};

Modem.prototype.dial = function (options, callback) {
  var data;
  var self = this;

  var optionsf = {
    path: options.path,
    method: options.method,
    headers: options.headers || Object.assign({}, self.headers),
  };

  optionsf.headers['Content-Type'] = 'application/json';

  if (options.data) {
    data = JSON.stringify(options.data);
    optionsf.headers['Content-Length'] = Buffer.byteLength(data);
  }

  optionsf.socketPath = this.socketPath;

  this.buildRequest(optionsf, options, data, callback);
};

Modem.prototype.buildRequest = function (options, context, data, callback) {
  var self = this;
  var connectionTimeoutTimer;

  var req = http.request(options, function () { });

  debug('Sending: %s', util.inspect(options, {
    showHidden: true,
    depth: null
  }));
```

```
if (self.connectionTimeout) {
  connectionTimeoutTimer = setTimeout(function () {
    debug('Connection Timeout of %s ms exceeded', self.connectionTimeout);
    req.abort();
  }, self.connectionTimeout);
}

if (self.timeout) {
  req.on('socket', function (socket) {
    socket.setTimeout(self.timeout);
    socket.on('timeout', function () {
      debug('Timeout of %s ms exceeded', self.timeout);
      req.abort();
    });
  });
}

req.on('connect', function () {
  clearTimeout(connectionTimeoutTimer);
});

req.on('disconnect', function () {
  clearTimeout(connectionTimeoutTimer);
});

req.on('response', function (res) {
  clearTimeout(connectionTimeoutTimer);

  var chunks = [];
  res.on('data', function (chunk) {
    chunks.push(chunk);
  });

  res.on('end', function () {
    var buffer = Buffer.concat(chunks);
    var result = buffer.toString();

    debug('Received: %s', result);

    var json;
    try {
      json = JSON.parse(result);
    } catch (e) {
      json = null;
    }
    self.buildPayload(null, context.statusCodes, res, json, callback);
  });
});

req.on('error', function (error) {
```

```
    clearTimeout(connectionTimeoutTimer);
    self.buildPayload(error, context.statusCodes, {}, null, callback);
  });

  if(data) {
    req.write(data);
  }
  req.end();
};

Modem.prototype.buildPayload = function (err, statusCodes, res, json, cb) {
  if (err) return cb(err, null);

  if (statusCodes[res.statusCode] !== true) {
    var msg = new Error(
      '(HTTP code ' + res.statusCode + ') ' +
      (statusCodes[res.statusCode] || 'unexpected') + ' - ' +
      (json.fault_message || json) + ' '
    );
    msg.reason = statusCodes[res.statusCode];
    msg.statusCode = res.statusCode;
    msg.json = json;
    cb(msg, null);
  } else {
    cb(null, json);
  }
};


module.exports = Modem;
```

Додаток 3

Копія презентації

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО”

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

КАФЕДРА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ КОМП’ЮТЕРНИХ СИСТЕМ

АЛГОРИТМІЧНО-ПРОГРАМНИЙ МЕТОД ПЛАНУВАННЯ ТА ДИНАМІЧНОГО ВИДІЛЕННЯ РЕСУРСІВ ДЛЯ ХМАРНИХ ОБЧИСЛЕНЬ

Доповідач: Кривенко Петро Олегович

Науковий керівник: к.т.н., ст. викладач Хіцко Я.В.

Київ – 2020

АКТУАЛЬНІСТЬ ДОСЛІДЖЕННЯ

- Хмарні обчислення є однією з найважливіших технологій сучасності. Оскільки ця технологія обслуговує мільйони користувачів, явно постала проблема оптимального виділення ресурсів
- Ріст популярності сервісів для надання доступу до ресусів на вимогу

Об’єкт дослідження: процес планування задач та виділення ресурсів у хмарних обчисленнях.

Предмет дослідження: методи планування задач та виділення ресурсів у хмарних обчисленнях.

ЗАВДАННЯ ТА МЕТА ДОСЛІДЖЕННЯ

Наукове завдання: розробити метод планування задач та динамічного виділення ресурсів у хмарних обчисленнях.

Мета дослідження: оптимізація обробки запитів на виконання задач у системах хмарних обчислень.

ОКРЕМІ ЗАВДАННЯ

1. Провести аналіз існуючих алгоритмів планування та виділення ресурсів CPU у хмарних обчисленнях.
2. Розробити гібридний алгоритм планування задач та динамічного виділення ресурсів CPU.
3. Порівняти існуючі алгоритми з запропонованим гібридним алгоритмом.
4. Розробити метод на основі запропонованого гібридного алгоритму планування задач та динамічного виділення ресурсів.
5. Розробити систему обробки запитів на виконання задач.
6. Провести аналіз отриманих результатів.

НАЙБІЛЬШ ВІДОМІ АЛГОРИТМИ ПЛАНУВАННЯ ЗАДАЧ

- First Come First Serve (FCFS)
- Round Robin (RR)
- Shortest Job First (SJF)

НАЙБІЛЬШ ВІДОМІ АЛГОРИТМИ ПЛАНУВАННЯ ЗАДАЧ

First Come First Serve - порядок задач у черзі оснований на часі додавання до черги. Задача що прийшла першою буде першою відправленна на обробку.

Переваги:

- Популярний та простий у реалізації алгоритм
- FIFO порядок обробки задач
- Є “справедливим” до задач

Недоліки:

- Високий час очікування задач у черзі
- Ресурси не використовуються оптимально

НАЙБІЛЬШ ВІДОМІ АЛГОРИТМИ ПЛАНУВАННЯ ЗАДАЧ

Shortest Job First - задачі відсортовані за пріорітетом, який надається на основі часу необхідного для виконання. Чим меншою є задача тим більший вона матиме пріорітет.

Переваги:

- Середній час очікування є мінімальним серед усіх агоритмів

Недоліки:

- Не є справедливим до ресурсоємних задач

НАЙБІЛЬШ ВІДОМІ АЛГОРИТМИ ПЛАНУВАННЯ ЗАДАЧ

Shortest Job First - задачі відсортовані за пріорітетом, який надається на основі часу необхідного для виконання. Чим меншою є задача тим більший вона матиме пріорітет.

Переваги:

- Середній час очікування є мінімальним серед усіх агоритмів

Недоліки:

- Не є справедливим до ресурсоємних задач

НАЙБІЛЬШ ВІДОМІ АЛГОРИТМИ ПЛАНУВАННЯ ЗАДАЧ

Round Robin - алгоритм який циклічно обробляє усі наявні задачі у черзі, виділяючи усім однаковий квантум часу на виконання.

Переваги:

- Зрозумілий та простий у реалізації
- Однаково справедливий до усіх задач

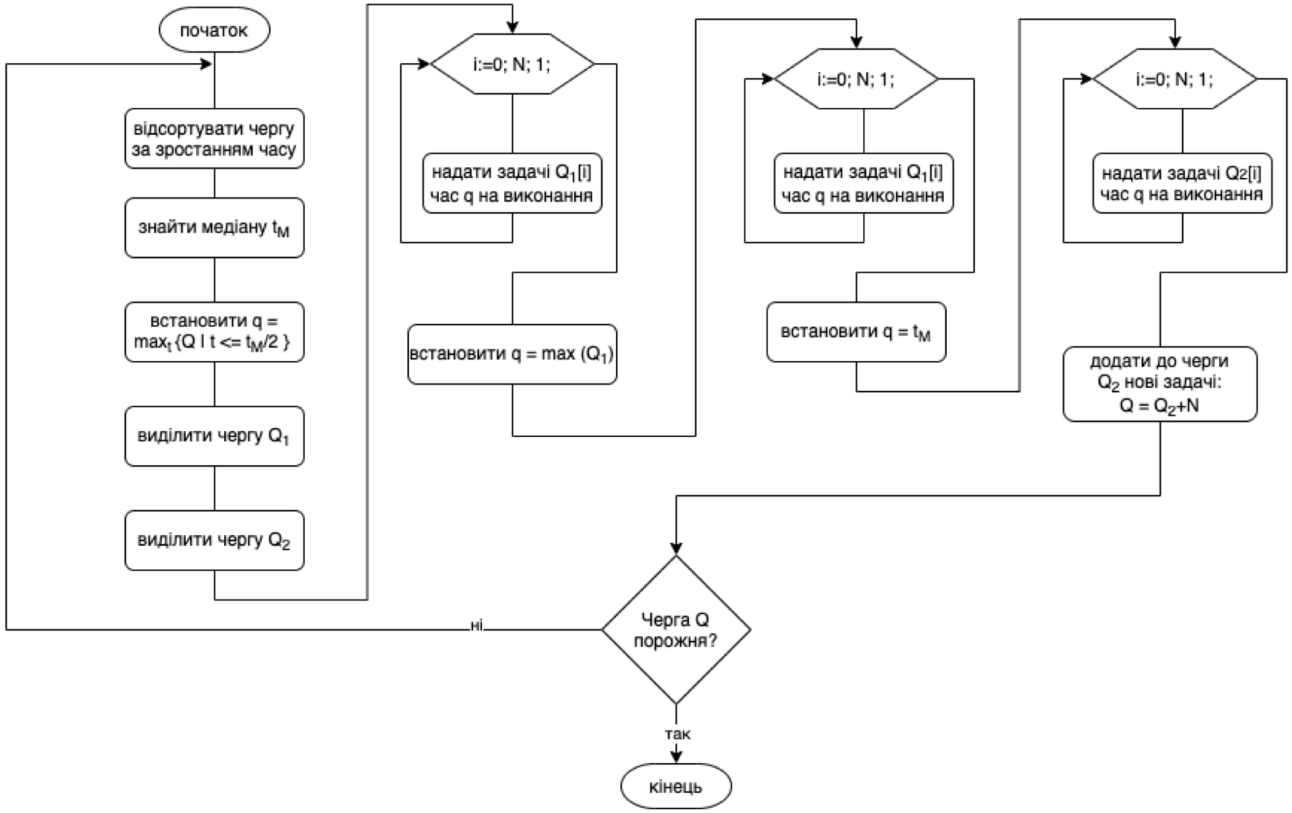
Недоліки:

- Вибір квантуму часу є складною задачею

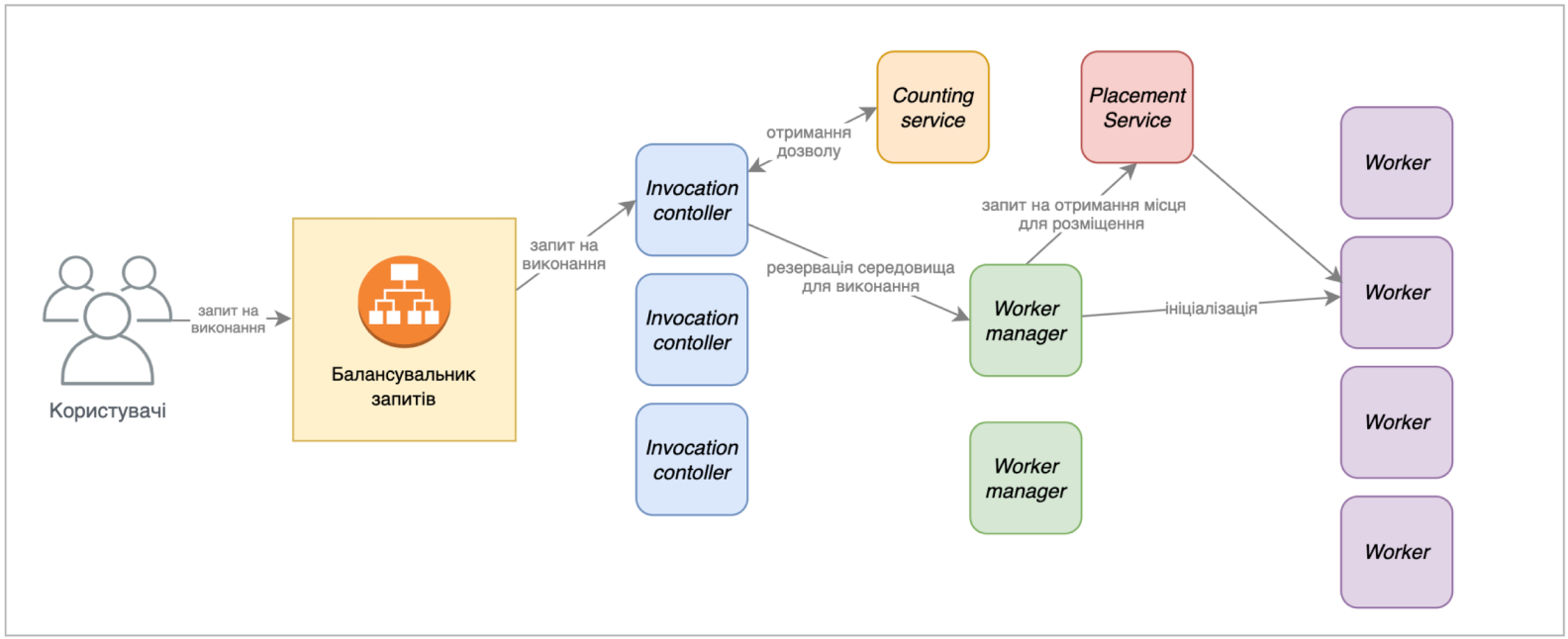
ПОРІВНЯННЯ ІСНУЮЧИХ АЛГОРИТМІВ

	Проблема голодування	Пропускна здатність	Час до завершення	Час відповіді
First Come First Serve	+/–	Низька	Швидкий	Повільний
Shortest Job First	+	Висока	Середній	Середній
Round Robin scheduling	–	Середня	Середній	Швидкий

ЗАПРОПОНОВАНИЙ АЛГОРИТМ ПЛАНУВАННЯ ЗАДАЧ (ГІБРИДНИЙ АЛГОРИТМ SJF + RR)

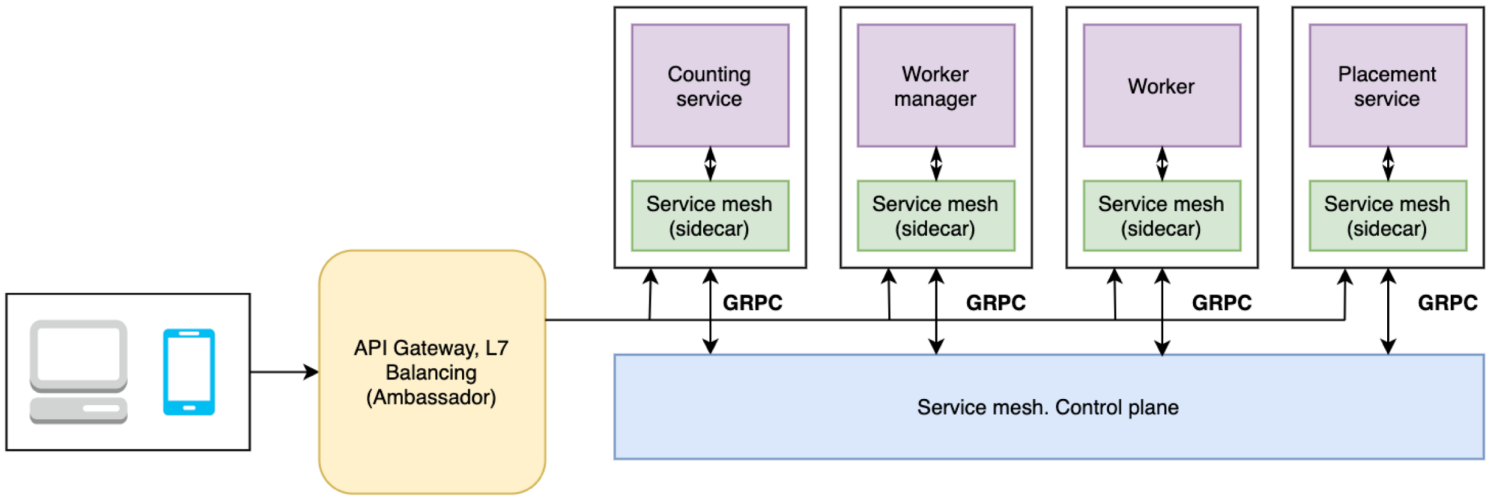


АРХІТЕКТУРА РОЗРОБЛЕНОЇ СИСТЕМИ



ОБРАНІ ТЕХНОЛОГІЇ

ВЗАЄМОДІЯ СЕРВІСІВ СИСТЕМИ



Опис сервісів системи. Counting service.

Обов’язки сервісу:

- Моніторинг рівня завантаженості системи
- Контроль за масштабуванням

```
service CountingSevice {
  rpc checkConcurrency(CheckConcurrencyRequest) returns (CheckConcurrencyResponse) {}
  rpc notify(ChangeConcurrencyRequest) returns (ChangeConcurrencyResponse) {}
}
```

Опис сервісів системи. Worker service

Обов’язки сервісу:

- Створення та керування віртуальними машинами
- Забезпечення ізоляції контексту
- Віртуалізація та емуляція пристроїв
- Надання програмногой інтерфейс для взаємодії з сервісом

```
service WorkerService {
  rpc createVM(CreateVMRequest) returns (CreateVMResponse) {}
  rpc makeSnapshot(MakeSnapshotRequest) returns (MakeSnapshotResponse) {}
  rpc restoreFromSnapshot(RestoreSnapshotRequest) returns (RestoreSnapshotResponse) {}
}
```

Опис сервісів системи. Worker manager

Обов’язки сервісу:

- Обробка черги задач
- Керування процесом виконання задачі
- Запит на виділення нових ресурсів

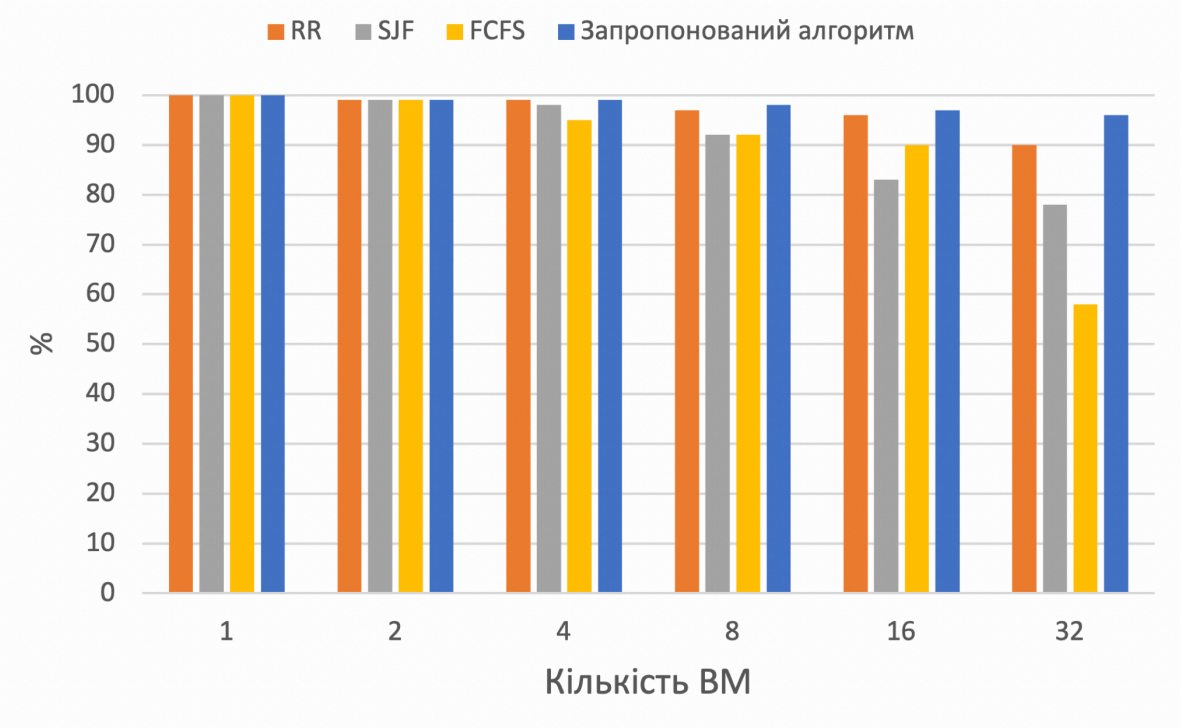
Опис сервісів системи. Placement service

Обов'язки сервісу:

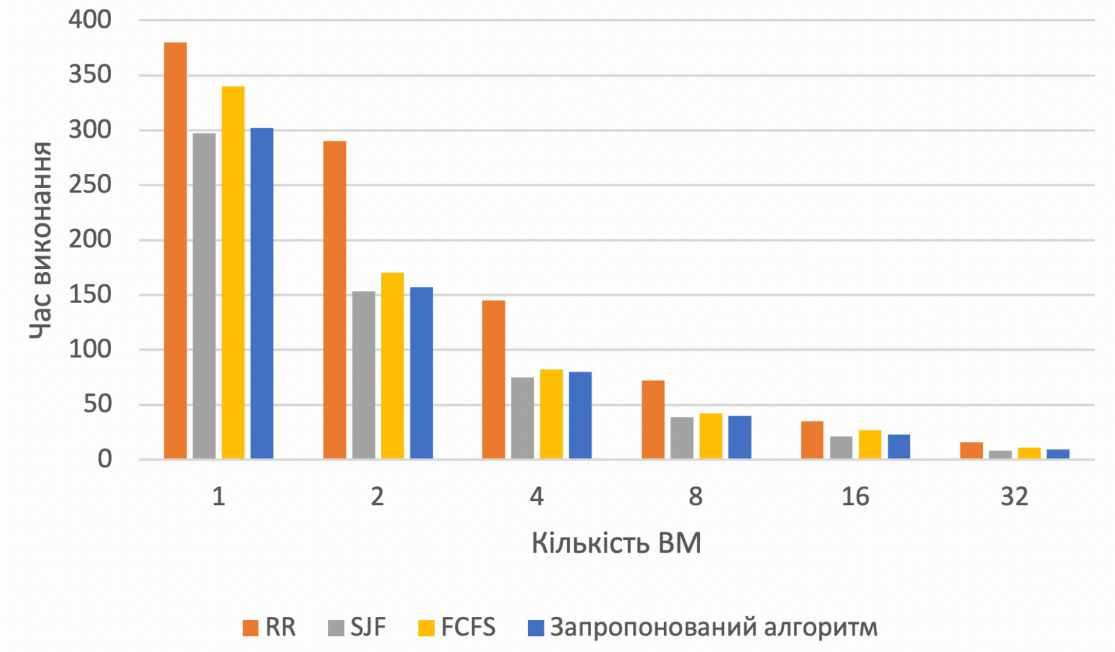
- Розміщення нових віртуальних машин
- Надання пограмного інтерфейсу для отримання віртуальної машини

```
service PlacementService {
  rpc getVM(GetVMRequest) returns (GetVMResponse) {}
}
```

Тестування алгоритму. Завантаженість CPU



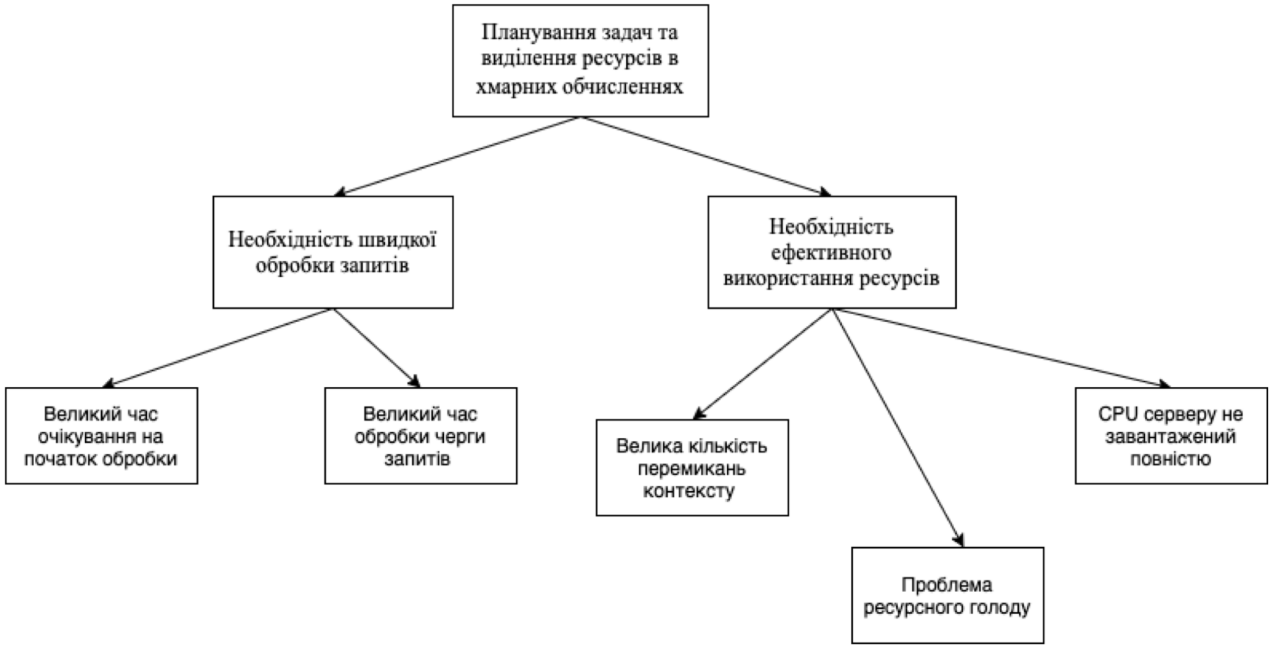
Тестування алгоритму. Час обробки черги



Тестування системи

```
➜  diploma hey -t 50  -c 5 -n 1000 http://localhost/hello

Summary:
  Total:        1125.8082 secs
  Slowest:      13.7845 secs
  Fastest:      0.0315 secs
  Average:      5.3493 secs
  Requests/sec: 0.8883

  Total data:   18150 bytes
  Size/request: 18 bytes

Response time histogram:
  0.031 [1]     |
  1.407 [47]    |
  2.782 [186]   |
  4.157 [179]   |
  5.533 [128]   |
  6.908 [134]   |
  8.283 [150]   |
  9.659 [100]   |
  11.034 [52]   |
  12.409 [18]   |
  13.784 [5]    |


Latency distribution:
  10% in 1.8769 secs
  25% in 2.8683 secs
  50% in 5.0517 secs
  75% in 7.6428 secs
  90% in 9.3433 secs
  95% in 10.0912 secs
  99% in 12.0430 secs

Details (average, fastest, slowest):
  DNS+dialup:   0.0000 secs, 0.0315 secs, 13.7845 secs
  DNS-lookup:   0.0000 secs, 0.0000 secs, 0.0024 secs
  req write:    0.0000 secs, 0.0000 secs, 0.0009 secs
  resp wait:    5.3492 secs, 0.0314 secs, 13.7844 secs
  resp read:    0.0001 secs, 0.0000 secs, 0.0005 secs
```

Дерево проблем



Канва бізнес-моделі

<table>
<tr>
<td rowspan="2">Проблема
неефективне використання обчислювальних ресурсів;
складність у налаштуванні та підтримці серверів;
для одноразових обчислень придбати сервери не оптимально.</td>
<td>Рішення
Хмарний обчислювальний сервіс з більш оптимальним способом використання ресурсів.</td>
<td>Унікальна ціннісна пропозиція
Розроблена платформа для хмарних обчислень з веб-інтерфейсом, яка дає можливість економити кошти завдяки покращеному способу балансування навантажень та виділення ресурсів.</td>
<td rowspan="2">Споживачі
аутсорсингові компанії;
розробники;
дослідники.</td>
</tr>
<tr>
<td>Ключові метрики
Сплачений час розрахунків.
Сплачений обсяг сховища даних.</td>
<td>Канали
Власна платіжна система у веб-додатку.</td>
</tr>
<tr>
<td colspan="2">Структура витрат
заробітна плата робітникам;
придбання апаратного забезпечення;
витрати на електроенергію та оренду приміщення.</td>
<td colspan="2">Потоки доходів
Продаж кожної мілісекунди активного навантаження ресурсу (але не менше 150 мс). Продаж кожного Мегабайту сховища даних.</td>
</tr>
</table>

НАУКОВА НОВИЗНА

Запропоновано метод планування задач та виділення ресурсів в хмарних обчисленнях, на основі гібридного алгоритму, який відрізняється від відомих одночасним застосуванням алгоритмів Round Robin та Shortest Job First. Це дозволяє зменшити час обробки завдань на 10-17% та підвищити ефективність використання ресурсів CPU на 23%.

ВИСНОВКИ

1. Проведено аналітичний огляд технологій хмарних обчислень, їх загальні характеристики, особливості, структуру та архітектуру та показано, що на сьогоднішній день залишається невирішеним питання виділення ресурсів та планування задач у хмарних обчисленнях.
2. Запропоновано та реалізовано модифікований метод планування задач та виділення ресурсів в хмарних обчисленнях, який відрізняється від відомих способів одночасним застосуванням алгоритмів Round Robin та Shortest Job First, що дозволяє зменшити час обробки завдань.
3. Проведено експериментальні дослідження, які показали, що час обробки завдань при використанні гібридного алгоритму (RR + SJF) зменшується на 10-17%.

Дякую за увагу!